1988

Design and construction of an IBM PC emulator for the Commodore Amiga 1000 /

Robert Thomas Krebbs
Lehigh University

DESIGN AND CONSTRUCTION OF AN IBM PC EMULATOR FOR THE COMMODORE AMIGA 1000

by

Robert Thomas Krebs

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science and Electrical Engineering

Lehigh University

1988

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Sept 19, 1988

Professor in Charge

Chairman of Department

Table of Contents

List of Figures

List of Schematics

Abstract

The design and construction of a hardware-based IBM PC emulator for the Amiga 1000 are discussed in this thesis. The emulator enables the Amiga 1000, a non IBM PC compatible computer, to run IBM PC software. The hardware components of the emulator, including a 128K block of shared memory which acts as the communication link between the emulator and the Amiga, are examined in detail. Software which provides the emulator with console I/O using the Amiga keyboard and video display is also discussed.

Chapter 1: Introduction

With the large amount of IBM PC software available today, a device which enables a non IBM PC compatible computer to run IBM PC software would be very useful. Such a device would allow the user of a non IBM PC compatible computer to take advantage of the many application programs available for the IBM PC, and still use programs written specifically for his particular system. Devices which perform this function are known are in existence, and are known as *emulators*.

The word "emulation" has many definitions. In the context of this thesis, an appropriate definition would be the duplication or imitation of the behavior of another computer. An IBM PC emulator would reproduce the operation of an IBM PC on a *host* machine (computer system in which emulator is being used). With the emulator operating, the host machine would behave indistinguishably from an IBM PC, and it could run software compiled to run on an IBM PC.

Emulators can be implemented in software or in hardware. In either case, the emulator must provide equivalents for the CPU in the *target* (emulated) machine and any ROMs specific to that system. A software-based emulator uses software to emulate both the CPU and the ROMs. An emulator program takes

instructions from a target system program and translates them into instructions compatible with the hardware present in the host system. A hardware-based emulator, in contrast, physically uses the same CPU as does the target system. This processor runs target system software while the CPU in the host system manages I/O.

Software emulators operate by providing translations for any machine instruction that may appear in a target system program. For every machine instruction, the CPU in the host system executes a subroutine written in compatible machine language. The subroutine performs the same operation as the target system machine instruction. A translation table is often used to contain the addresses of the subroutines. The target system machine instruction is used as an index into the table.

Software-based emulators store memory operands used by target system software in host system memory. The registers of the emulated CPU are usually mapped into the registers of the host CPU to achieve faster performance.

Hardware-based emulators run target system software by providing the same hardware as that found in the target system. The emulator uses the same CPU, or a functionally equivalent CPU, to run the software. While running, the target system software is stored in

private RAM in the emulator. To increase compatibility, the emulator may also use several of the same peripheral chips as the target system. In many respects, software runs on a hardware-based emulator exactly as it would on the target system.

When a hardware-based emulator is operating, its CPU usually acts as the main CPU in the host-emulator system. The CPU in the host machine acts as a peripheral I/O processor. In the Amiga 1000, however, multitasking allows the 68000 microprocessor in the Amiga to run Amiga software concurrently with handling I/O requests from an emulator.

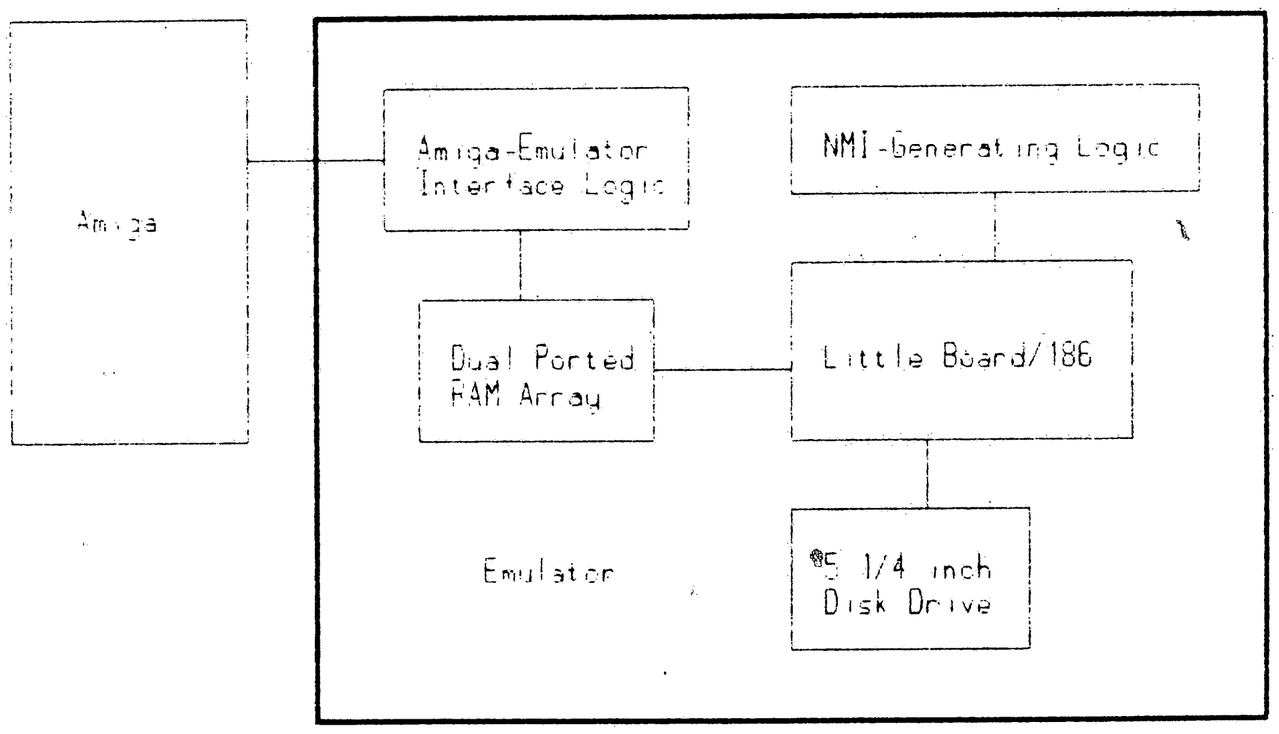
When comparing the performance and cost of software-based emulators vs. hardware-based emulators, differences are very apparent. Hardware-based emulators run 3 to 10 times faster than software-based emulators. When using a software-based emulator, the host CPU may have to execute several of its own machine instructions for every one target system machine instruction. Hardware-based emulators execute only the instructions that are actually in the target system software. Software-based emulators, however, are easier and quicker to develop. As a result, they are less expensive to purchase.

The bulk of this thesis focuses on the design and construction of a hardware-based IBM PC emulator for the

Amiga 1000. The design is introduced in Chapter 2. The theory of operation of the hardware and the software is explained in Chapters 3 and 4, respectively. Construction details are covered in Chapter 5. Chapter 6 is an evaluation of the design. Finally, the practicality of emulators is addressed in Chapter 7.

Chapter 2: Overview of Design

All hardware-based emulators must contain solutions for a common set of problems. An emulator must provide an environment (CPU, RAM, ROM) in which target system software can run. The transfer of data, particularly keyboard and video display data, between the emulator and the host must be facilitated by some form of communication link. Finally, a method for transporting target system software to the emulator must be established. Possible solutions to these problems are outlined in this chapter. These solutions form the basis for an emulator design, illustrated in Figure 2.1.



Block Diagram of Emulator

Figure 2.1

Since the CPU in the IBM PC is a 5 Mhz Intel 8088, any hardware-based IBM PC emulator must also be designed

around the 8088, or a close relative. The 8088, however, is rather obsolete because its data bus is only 8-bits wide. The Intel 80286 and 80386 offer significantly better performance, but are expensive. The Intel 8086 and 80186 are cheaper alternatives. Both have a 16-bit data bus, and can be found in 8 Mhz versions.

The amount of RAM to include in an IBM PC emulator should be based on the memory requirements of typical IBM PC software. Many programs require at least 256K of RAM. To be useful, an IBM PC emulator should have 512K of RAM. It should be noted that the original IBM PC had only 64K of RAM on the motherboard, although it could be expanded to 640K with expansion boards. Memory requirements have increased dramatically in just a few years.

The IBM PC also has bootstrap and BIOS (Basic Input/Output Service) code stored in ROM. The bootstrap code loads the disk-based operating system (MS-DOS or PC-DOS), and the BIOS code provides support for the operating system and applications programs. An emulator must provide equivalent ROM-based code.

For large designs, such as an IBM PC emulator, overall construction and debugging time can be reduced by incorporating a single board computer having the required CPU, RAM, and ROM into the design. The single

board computer used in this design is the AMPRO Little Board/186. This computer has an 8 Mhz 80186, 512K of dynamic RAM, and ROM code compatible with PC-DOS. The Little Board/186 is designed specifically to be used in embedded applications, and is not a stand alone machine.

The communication link between the emulator and the Amiga is a 128K dual ported RAM array. Both the 80186 in the Little Board/186 and the 68000 in the Amiga can write data to and read data from the RAM array. In many ways, the shared RAM acts as a mailbox. One CPU can write data into the shared RAM, the other CPU can read it, and, perhaps, write back a response.

In the current design, two kinds of data are transferred via the shared RAM: video data and keyboard data. In the IBM PC, a portion of the memory address space (A0000h - BFFFFh) is set aside for storage of video data. Information to be displayed on the monitor is written into this area of memory address space (typically known as "video RAM"). To duplicate this operation with the emulator, the 128K shared RAM is mapped into 80186 memory space starting at address A0000h. IBM PC software running on the emulator can write display data to this area of memory space just as it can while running on the IBM PC. At periodic intervals, the Amiga reads the video data in the shared RAM, and updates its display.

The mapping of the shared RAM into the space normally occupied by video RAM in the IBM PC does not present a problem in this design. The Little Board/186 is designed to be used with a serial ASCII terminal, and does not have any video RAM. When the Little Board/186 is used "as is", all keyboard input and video output is done through a serial port.

Keyboard data is handled in a similiar manner. Codes for keypresses on the Amiga keyboard are written into the shared RAM. The 80186 reads the keyboard data, and processes it.

A straightforward approach is employed to transport IBM PC software to the emulator. A 5¼ IBM PC compatible disk drive is connected directly to the Little Board/186. The disk drive allows the emulator to access software and data stored on 360K 5¼ disks, a standard storage medium for IBM PC software.

A problem which should be addressed, but is not critical, is that of I/O port accesses. IBM PC software may attempt to directly access peripheral chips present in the IBM PC (all peripheral chips in the IBM PC are located in I/O address space). For example, some IBM PC programs access status registers in the Color Graphics Adapter (CGA) display card to determine when it is safe to access video RAM. If an IBM PC emulator does not have the same chips as the IBM PC, or if the

chips are at different addresses, some software may not run properly. An emulator should provide hardware and software which will handle I/O port accesses in an intelligent manner.

Hardware for handling I/O port accesses is present in the emulator design. Emulator logic generates a non maskable interrupt (NMI) whenever the 80186 attempts certain I/O operations. The NMI interrupt handler identifies the specific operation that caused the interrupt, and takes appropriate action.

Chapter 3: Theory of Operation - Hardware

This chapter is a more in depth description of the hardware components mentioned in the previous chapter. Most of the discussion is at a functional level. Lower level details such as timing are covered only where this information is necessary to understand the functional operation.

Throughout this chapter, it is assumed that the reader has a general knowledge of microprocessors and hardware interfacing techniques. Specific knowledge of the larger chips used in the design, however, is not necessary. Sources for additional information on these chips are mentioned at appropriate points in the chapter.

Amiga Expansion Bus and Amiga-Emulator Interface Logic

Hardware expansion of the Amiga 1000 is facilitated by an 86 pin expansion bus. The bus is accessible externally through a slot in the right side of the Amiga. A hardware device can be interfaced to the bus directly, or through an expansion chassis.

The Amiga 1000 expansion bus, illustrated in Figure 3.1, is essentially an extension of the 68000 microprocessor bus. Many of the signals on the bus are driven directly by the 68000 inside the Amiga. The

protocols for data transfer on the expansion bus are also based those of the 68000 bus.

GND	1	2	GND
GND	3	4	GND
NC	5	6	NC
DMAOUT*	7	8	NC
NC	9	10	NC
SLAVEOUT*	11	12	CONFIG_IN11*
GND	13	14	C3*
CDAC	15	16	C1*
OVR*	17	18	XRDY
INT2*	19	20	RESERV2
pA5	21	22	INT6*
pA6	23	24	pA4
GND	25	26	pA3
pA2	27	28	pA7
pA1	29	30	pA8
FC0	31	32	pA9
FC1	33	34	pA10
FC2	35	36	pA11
GND	37	38	pA12
pA13	39	40	IPL0*
pA14	41	42	IPL1*
pA15	43	44	IPL2*
pA16	45	46	BERR*
pA17	47	48	VPA*
GND	49	50	E
VMA*	51	52	pA18
RES*	53	54	pA19
HLT*	55	56	pA20
pA22	57	58	pA21
pA23	59	60	BR*
GND	61	62	BGACK*
pD15	63	64	BGIN*
pD14	65	66	DTACK*
pD13	67	68	pR/W*
pD12	69	70	LDS*
pD11	71	72	UDS*
GND	73	74	AS*
pD0	75	76	pD10
pD1	77	78	pD9
pD2	79	80	pD8
pD3	81	82	pD7
pD4	83	84	pD6
GND	85	85	pD5

Amiga Expansion Bus

Figure 3.1

Nearly half of the 86 pins on the expansion bus are allocated to the "23-bit" address bus and 16-bit data bus of the 68000. The 68000 address bus is rather unusual in that, while internal addresses are 24-bits

long, only 23 address bits (A23-A21) are available external to the chip. A0 is used internally to generate the data strobes (UDS* and LDS*). While the external address bus consists of only 23 bits, all memory addresses are actually 24-bits long and can access 16 megabytes of memory.

UDS* and LDS* indicate whether data is being transferred on the upper half, the lower half, or both halves of the 16-bit data bus. When UDS* is LOW, data is being transferred on the upper half of the data bus. LDS* is LOW during tranfers involving the lower half of the data bus.

The 68000 transfers even addressed bytes on D15-D8, and odd addressed bytes on D7-D0. This interface is exactly opposite that of the 8086 and 80186. These processors transfer even bytes on D7-D0, and odd addressed bytes on D15-D8.

The 68000 pulses the address strobe (AS*) LOW to indicate that a valid address is being output on the address bus. An active AS* also indicates that the function code being output on FC2, FC1, and FC0 is valid. The function code identifies the type of bus activity that is currently taking place.

The read/write signal (R/W*) is HIGH for all read cycles, and LOW for all write cycles.

The data transfer acknowledge signal (DTACK*)

must be asserted by external logic during every read and write cycle. The 68000 automatically inserts wait states in the bus cycle until DTACK* becomes active. DTACK* must be pulsed; it cannot be tied LOW.

The bidirectional RESET* signal serves a variety of functions. As an input, RESET* can be asserted in conjunction with HALT* to reset the 68000. When the 68000 executes a Reset instruction, RESET* is asserted as an output signal. When used in this way, RESET* can initialize other devices in the system without the 68000 being reset.

The bus arbitration signals (BR*, BG*, and BGACK*) and the interrupt request inputs (IPL2, IPL1, and IPL0) are not pertinent to the emulator design and will not be discussed. The signals VMA*, VPA*, and E*, used for interfacing the 68000 to 6800 class peripheral chips, are also not relevant to the design. More information on these signals can be found in the 68000 Microprocessor Handbook (see Bibliography).

Several other expansion bus signals, not directly related to the 68000, are also used in the emulator design. In Designing Hardware for the Amiga Expansion Architecture, Commodore recommends that DTACK* should not be asserted by external hardware. Internal Amiga hardware automatically asserts this signal. Instead, XRDY should be pulled LOW by external hardware devices

which require a longer bus cycle. A LOW level on XRDY will cause wait states to be inserted in the current bus cycle.

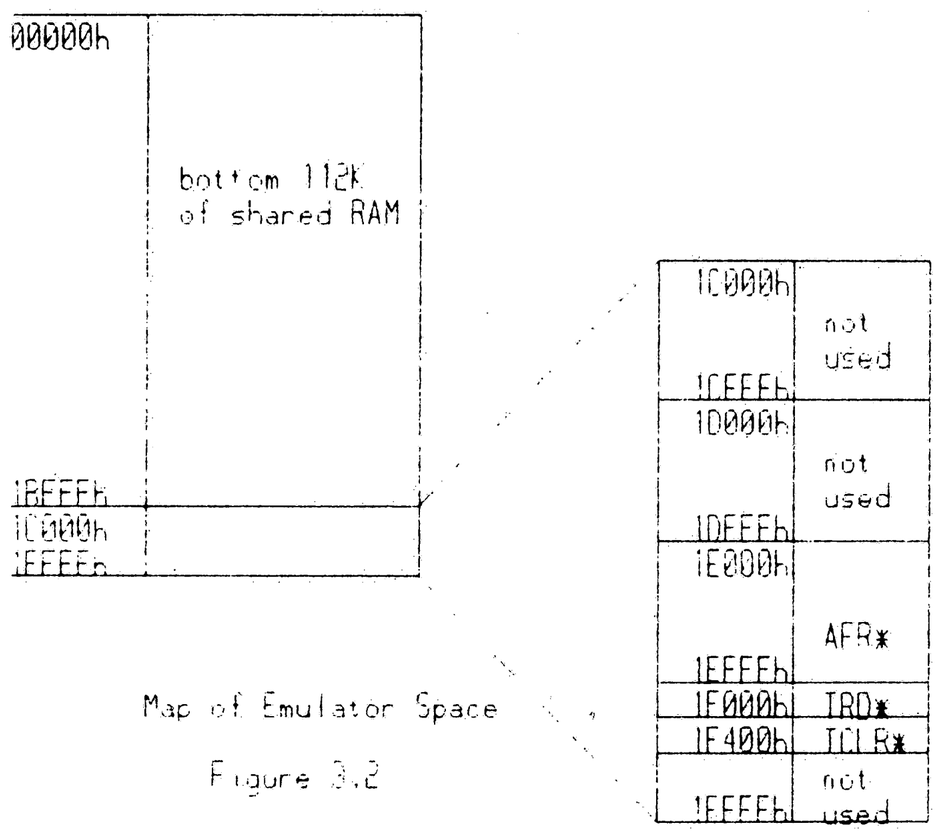
Three clock signals (C1*, C3*, and CDAC) are available on the expansion bus. C1* is the inverted internal 3.597 Mhz color clock. C3* is equivalent to C1* shifted 90 degrees. The 7.14 Mhz processor clock (7M), is not available on the expansion bus, but it can be generated using the equation: C3* XNOR C1* = 7Meqiv. CDAC is equivalent to 7M shifted 90 degrees.

The logic of the emulator that interfaces with the Amiga expansion bus is shown in Schematics 1a and 1b. This logic protects internal Amiga circuitry by buffering signals on the expansion bus. It also facilitates the mapping of the emulator into Amiga expansion space.

In Designing Hardware for the Amiga Expansion Architecture, Commodore recommends that the load on any signal on the expansion bus should not exceed 1 TTL "F" load. An typical "F" load is 1.25/1.25 TTL unit loads. To meet this requirement, 74LS245 transceivers are used to buffer the address bus, data bus, and various other signals.

The emulator occupies 128K of Amiga expansion space. As can be seen in Figure 3.2, most of the 128K

is taken up by the lower 112K of the dual ported RAM array. The remaining 16K is decoded by a 74LS139 decoder (U12) to produce three signals: TRD*, TCLR*, and AFR*. A 68000 access to the memory block associated with a particular signal will pulse that signal. The uses of these signals are discussed later in this chapter.



Map of Emulator Space

Figure 3.2

Switches are used to set the base address of the emulator, and, thereby, map the emulator into Amiga expansion space. Switches S6-S0 of SW1 are set to the complement of the 7 high order address bits (A23-A17) of the base address. In the current design, the Amiga auto-configuration protocol, which can automatically map hardware into the expansion space, is not supported.

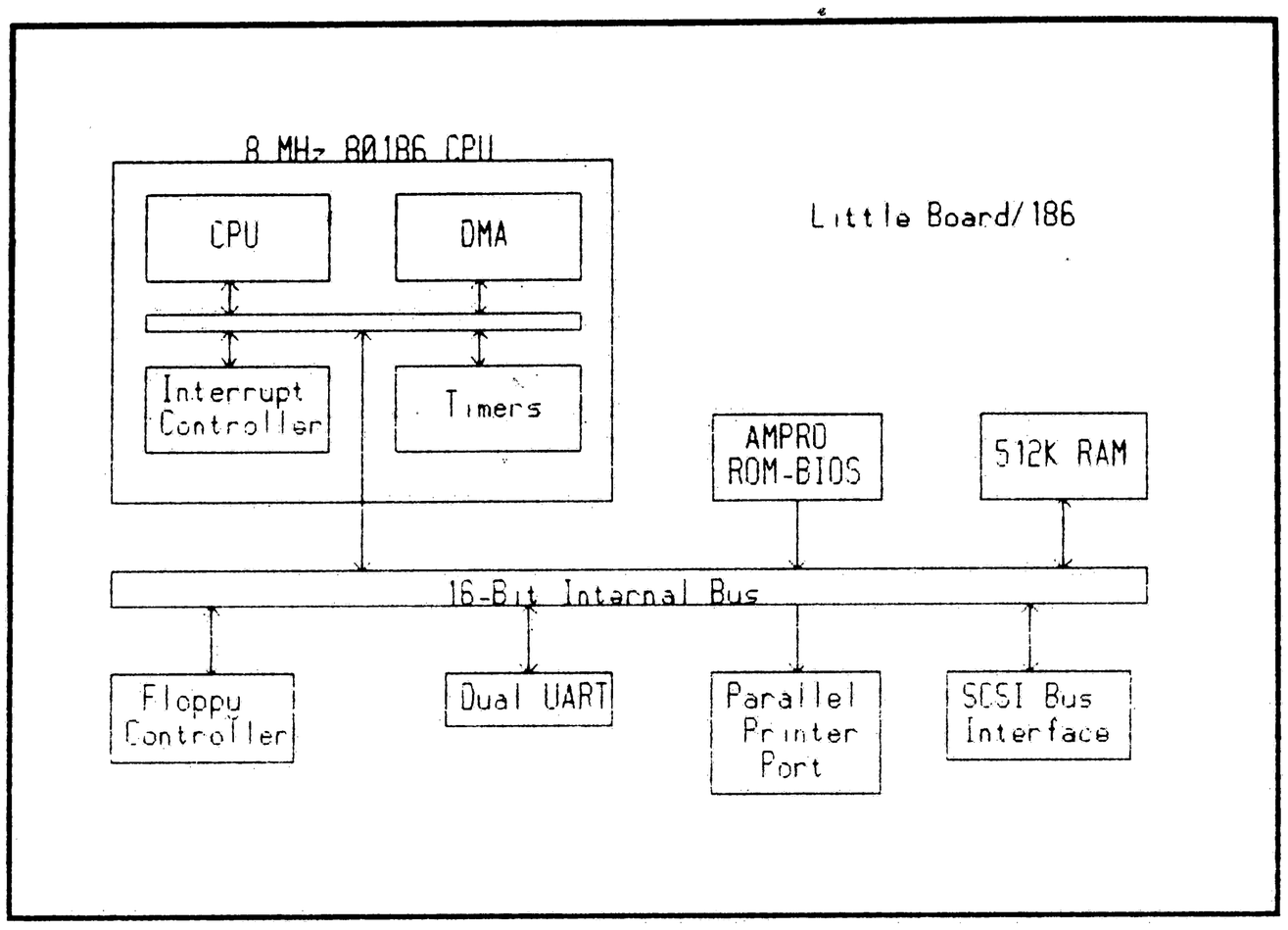
The emulator logic which pulls XRDY to a LOW level

is illustrated in Schematic 1b. This logic (U16.1) pulls XRDY low whenever the 68000 accesses the dual ported RAM array. Wait states in the bus cycle may be needed because the 68000 may not get immediate access to the shared RAM. The controller of the dual ported RAM array asserts AACKB* to indicate that the 68000 can complete the bus cycle.

Little Board/186 and Proto/186

The Little Board/186 (see Figure 3.3) is built around the Intel 80186 high integration microprocessor unit. The 80186 contains a 16-bit CPU , a 2-channel DMA controller, three 16-bit timers, and a programmable interrupt controller. It also has programmable chip select and wait state generation logic. By combining several normally separate components into one package, the 80186 uses less board space and power.

The CPU portion of the 80186 is essentially an enhanced 8086. The two processors share the same general register set, and the 8086 instruction set is a subset of the 80186 instruction set. Instruction execution times for the 80186, however, are shorter because hardware , rather than microcode, is used to calculate effective addresses. All object code which runs on the 8086 and 8088, including IBM PC software, should also run, and probably run faster, on the 80186.



Block Diagram of Little Board/186

Figure 3.3

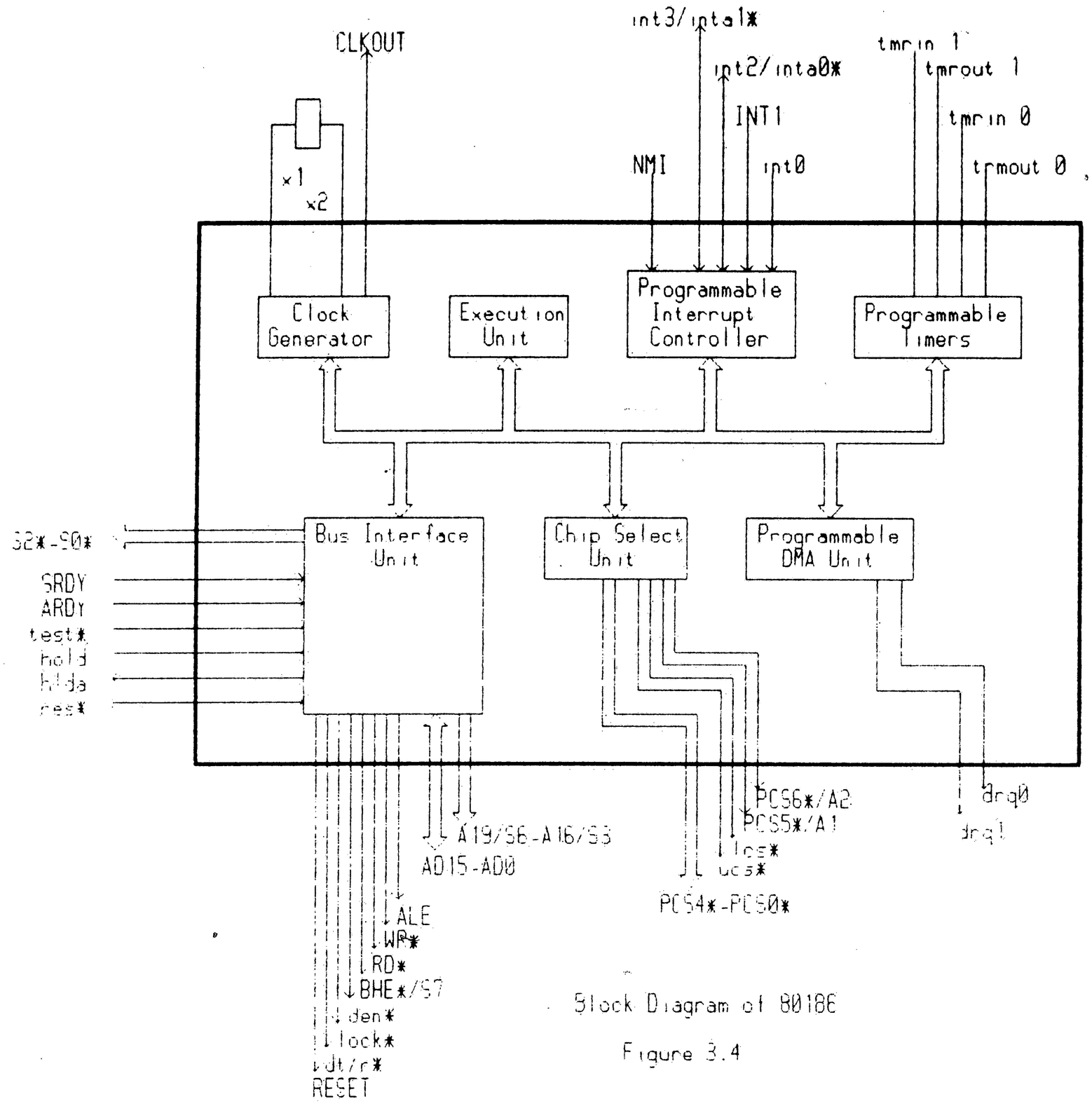
The integrated peripheral components (timers, DMA controller, interrupt controller, etc.) of the 80186 perform many of the same functions on the Little Board/186 as do the separate peripheral chips (8253, 8237, 8259) on the IBM PC motherboard. The interrupt controller receives and arbitrates between requests from four sources: the 2681 serial controller, the 5380 SCSI interface controller, the 1772 floppy disk controller, and the Centronics printer port. DMA channel 1 is used to refresh the 512k of DRAM (occupying the lower half

of the memory address space) on the Little Board/186. It is programmed to generate 20-bit addresses, timed by timer channel 2, with no terminal count. DMA channel 2 is shared between the floppy disk and the SCSI interface. Timer channel 2 acts as a prescaler to channel 0, and also provides the required 16 microsecond DRAM refresh clock. The external counter input of timer channel 1 is driven by the timer output of the 2681, allowing the timing of long intervals.

A block diagram of the 80186 appear in Figure 3.4. A basic knowledge of the signals appearing in capital letters in Figure 3.4 is helpful in understanding the overall functioning of the emulator. These signals will be discussed further. The signals appearing in small letters are used only locally on the Little Board/186, or are not used at all. The pin definitions for these signals, and timing information for all the signals, can be found in the iAPX 86/88 186/188 User's Manual, Hardware Reference and the Little Board/186 Technical Manual.

The address/data bus signals (AD15-AD0) make up the time multiplexed address/data bus. During the first clock period (T1) of the bus cycle, the 80186 asserts a memory or I/O port address on AD15-AD0. During the second clock period (T2), the 80186 removes the address from the bus. AD15-AD0 are then tristated for a

read cycle, or driven with data for a write cycle.



Block Diagram of 80186

Figure 3.4

The address latch enable (ALE) is provided to latch the address on AD15-AD0 into 8282/8263 or 74373 latches. ALE is active HIGH, and can directly drive the latch enable inputs of the latches. Addresses are guaranteed to be active on the falling edge of ALE.

The four most significant address bits are asserted on A19/S6, A18/S5, A17/S4, and A16/S3 during T1. For the rest of the bus cycle, status information is

available on these lines. S6 is HIGH for processor cycles, and LOW for DMA cycles. S5-S3 are defined as LOW after T1.

The three least significant status bits are output on S2*, S1*, and S0*. The status bits are encoded as in Figure 3.5.

80186 Bus Cycle Status Information			
S2*	S1*	S0*	Bus Cycle
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	Halt
1	0	0	Instruction Fetch
1	0	1	Read from Memory
1	1	0	Write to Memory
1	1	1	Passive

Encoding of Status Bits

Figure 3.5

The bus high enable signal (BHE*) is low during T1 of read, write, and interrupt acknowledge bus cycles in which a byte is to be transferred on the upper half of the data bus. A0 is low during T1 of bus cycles in which a byte is to be transferred on the lower half of the data bus.

The read strobe (RD*) indicates that the 80186 is performing a memory or I/O read bus cycle. RD* is active LOW during T2, T3, and Tw (wait states) of any read cycle.

The write strobe (WR*) indicates that the data on the bus is to be written into a memory or I/O device.

WR* is active LOW during T2, T3, and Tw of any write cycle.

The non maskable interrupt signal (NMI) is an edge triggered input. A transition from low to high on this pin initiates a priority 1 (highest priority) interrupt at the next instruction boundary. NMI cannot be masked internally.

The maskable interrupt request inputs (INT3, INT2, INT1, and INT0) are active HIGH, and are programmed by AMPRO ROM-BIOS to be edge triggered. The default priorities for INT3, INT2, INT1, and INT0 are 9, 8, 7, and 6, respectively.

The asynchronous ready (ARDY) and synchronous ready (SRDY) inputs are used to inform the 80186 that the addressed memory or I/O device is ready to complete the bus cycle. Both inputs are active HIGH, but only the leading edge transition of ARDY is synchronized to the 80186 clock internally. Wait states are inserted in the bus cycle as long as both ARDY and SRDY remain LOW. Only one of these inputs must be active to terminate a bus cycle.

The peripheral chip select signals (PCS5*-PCS0*) are asserted during accesses to I/O space. The address ranges activating PCS5*-PCS0* are programmed by AMPRO ROM-BIOS.

The clock output signal (CLKOUT) provides the

system with a 50% duty cycle clock. All 80186 timing is relative to this clock. In the Little Board/186, the frequency of CLKOUT is 8 Mhz.

The reset output signal (RESET) indicates that the 80186 is being reset, and can be used a system reset. RESET is active HIGH, and is synchronized with the processor clock.

The AMPRO Proto/186 Prototype Adapter is also used in the emulator design. The Proto/186 facilitates construction of custom projects based on the Little Board/186. It provides approximately 18 square inches of wirewrap space, along with conditioned 80186 signals. The Proto/186 has the same dimensions and mounting slot positions as the Little Board/186, and is designed to plug directly into the 68-pin CPU header on the Little Board/186.

The logic present on the Proto/186 consists of one 74LS244 buffer, two 74LS245 transceivers, three 74F373 latches, and one control PAL. The address/data bus of the 80186 is buffered by the 74LS245 transceivers, generating the signals BAD15-BAD0. The transceivers are always enabled, and rely on the control PAL to switch direction. The latched address bus (LA19-LA0) is generated by the 74F373 latches. All 20 address bits are latched, as well S2*-S0* (LS2*-LS0*) and BHE*

(LBHE*). The 80186 signals RD*, WR*, ALE*, RESETOUT, and CLKOUT are buffered by the 74LS244 buffer, generating the signals BRD*, BWR*, BALE*, BRST, and BCLK. All other 80186 signals are also accessible, but are not buffered.

The control PAL also decodes the 80186 peripheral chip selects PCS0* and PCS6* into seven individual chip selects. Six of these chip selects are available for use on the Proto/186. The I/O port addresses assignments for the chip selects are shown in Figure 3.6.

Pal Pin	Signal Name	Address
18	PCS0A*	1000h-101Fh
17	PCS0B*	1020h-103Fh
16	PCS0C*	1040h-105Fh
15	PCS0D*	1060h-107Fh
14	PCS6A*	1300h-131Fh
13	PCS6B*	1320h-133Fh
12	PCS6C*	1340h-135Fh

Peripheral Chip Select Decoding

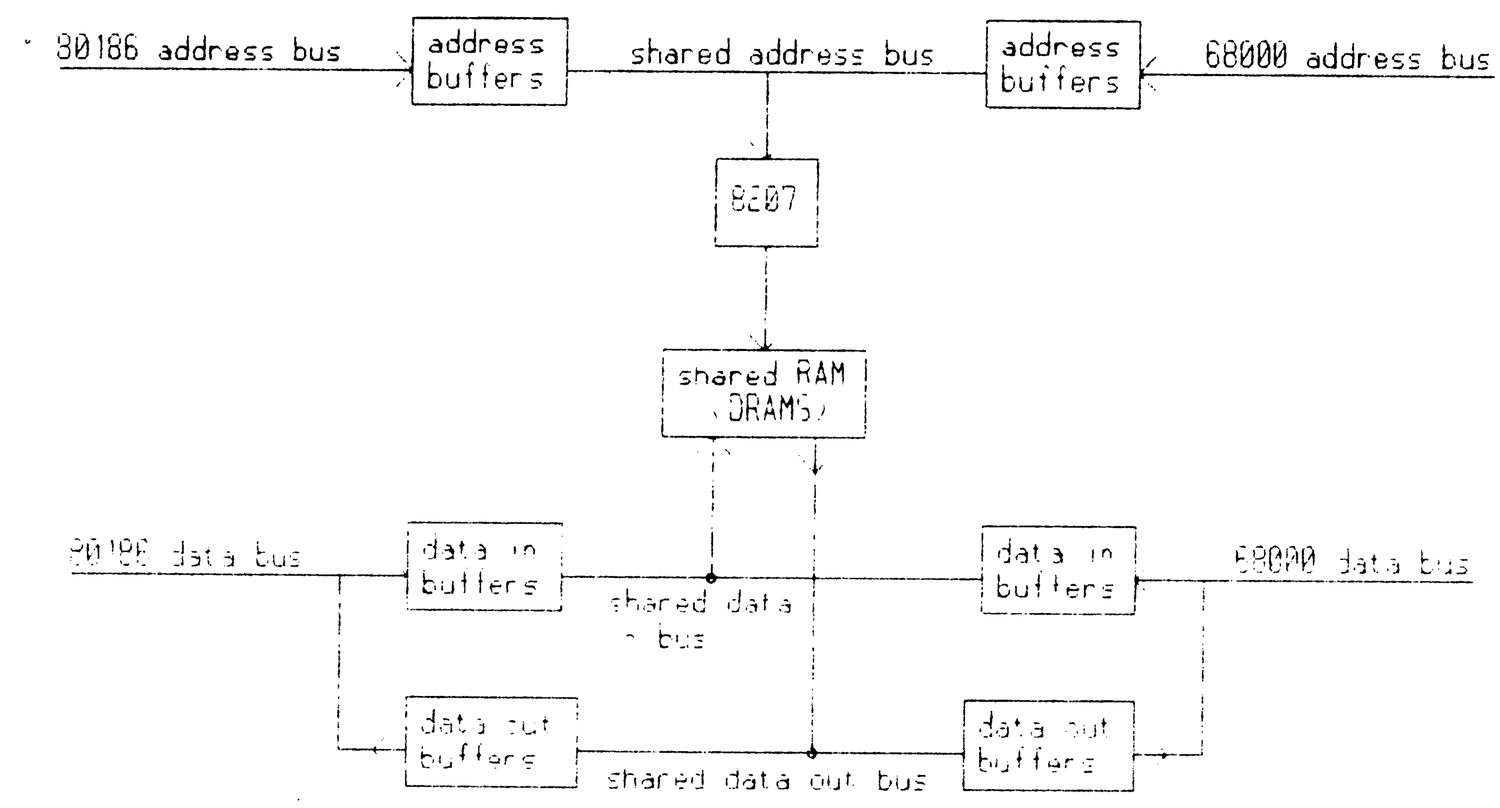
Figure 3.6

Dual Ported RAM Array

The major components of the dual ported RAM array are the Intel 8207 Dual Port Dynamic RAM Controller, two sets of address and data buffers, and the actual dynamic RAM chips. The relationship between these parts is illustrated in Figure 3.7. Detailed schematics of the

dual ported RAM array can be found in Appendix A.

It is assumed that the reader has a general knowledge of dynamic RAMs. Since the 4164 64K x 1 DRAMs used in this design have no unusual characteristics, the operation of these chips will not be discussed in detail.



Block Diagram of Dual Ported RAM array
Figure 3.7

The operation of the dual ported RAM array is controlled by the 8207. This chip provides all signals necessary to address and refresh 16K, 64K and 256K DRAMs. It also provides arbitration circuitry to allow two different processors to independently access the memory which it controls. Although the 8207 is optimized to run synchronously with the Intel 8086 family of microprocessors, it is capable of supporting the bus structures of other processors as well.

The arbitration circuitry of the 8207 receives access requests from three sources: Ports A, B, and C. In the emulator, Ports A and B are assigned to the 80186 and 68000, respectively. Port C is used by the internal refresh circuitry of the 8207.

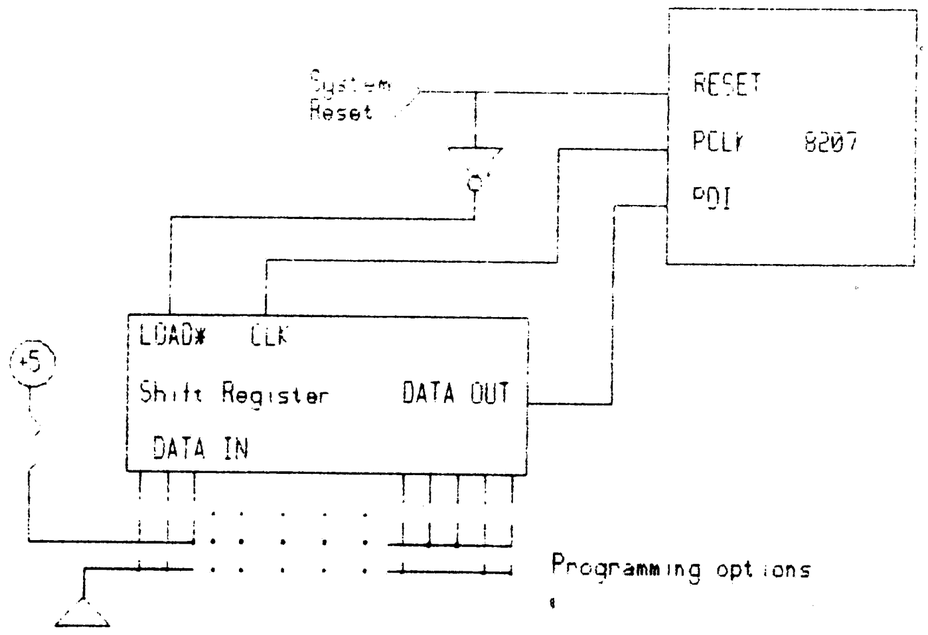
The operating characteristics of Ports A, B, and C are programmable. At the end of reset, the 8207 latches the states of PCTLA and PCTLB. If PCTL is high, the port is configured to be in status mode. In this mode, the status outputs (S2*, S1*, S0*) of an Intel 8086/186/286 microprocessor should drive the PCTL, RD*, and WR* input signals of that port. If PCTL is low, the port will be in command mode, and the RD* and WR* inputs should be driven the RD* and WR* signals of the CPU. In the emulator, PCTLA is pulled high with a pullup resistor, and PCTLB is tied permanently low.

The state of REF determines the refresh mode that the 8207 will use. If REF is tied high, as in the emulator design, the 8207 will generate internal refresh requests, and refresh the DRAM which it controls automatically.

It may appear that, for this design, automatic refreshing by the 8207 is redundant. The refresh operation performed by DMA channel 1 of the 80186 does cover the entire 1 megabyte memory address space, including the space occupied by the shared RAM. During

testing, however, it was determined that without the automatic refresh of the 8207, data is lost when the 68000 accesses the shared RAM continuously. It was concluded that the wait states caused by 68000 accesses slow down the DMA refresh cycles enough to cause loss of data. Tying REF HIGH solves this problem for the shared RAM, but the problem still exists for the rest of the RAM refreshed by DMA channel 1 (possible solutions to this problem are discussed in Chapter 6).

Additional programming options can be specified by using the logic illustrated in Figure 3.8. The reset pulse loads the program bits into the shift register. The 8207 then reads in the program bits by clocking MUX/PCLK sixteen times. The specific programming used for the emulator is shown in Figure 3.9, and is implemented with a 74LS165 shift register (U27). More details on programming the 8207 can be found in the Microprocessor and Peripheral Handbook, Volume 2.



Programming Shift Register

Figure 3.8

Whether initiated through Port A or through port B, all requests to access the shared RAM follow basically the same protocol. The 8207 queues a request from Port X when a read or write "command" on RDX*, WRX*, and PCTLX is qualified by an active PEX*. For Port A (status interface), the status signals driving PCTLA, RDA*, and WRA* must indicate a read or write bus cycle is occurring. PEA*, which is generated by address decoding logic, must also be active. For Port B (command interface), RDB* or WRB* must be active, along with PEB*.

Program Data Bit	Name	Programming for emulator
PD0	ECC	0 - non ECC mode
PD1	SA*	0 - Port A is synchronous
PD2	SB	0 - Port B is asynchronous
PD3	LFS*	1 - slow cycle 86 mode
PD4	RFS*	1 - slow RAM
PD5	RB0*	1
PD6	RB1*	1 - one bank of DRAMs
PD7	CI1	0 - 0% decrease in count
PD8	CI0	0 interval between refreshes
PD9	FLS*	0 - 128 rows in 2 ms refresh period
PD10	EXT	0 - no extended words
PD11	FFS*	0 - fast CPU frequency 0
PD12	PPR*	0 - most recently used Port has priority
PD13	TM1	0 - test mode off
PD14	0	0 - reserved, must be 0
PD15	0	0 - reserved, must be 0

8207 Programming for Emulator

Figure 3.9

The address decoding logic for Port A is a 74LS138 decoder (U29). The output signal, PrtAEn*, is active

for all memory read and write bus cycles which access the address range A0000h-BFFFFh. The address decoding logic for Port B has already been discussed.

The timing of PCTL, RD*, WR*, and PE* is critical. Because Port A is programmed as "synchronous", the 8207 requires that PCTLA, RDA*, WRA*, and PEA* be asserted such that they are all active at a particular rising clock edge. The timing requirements for Port B (asynchronous) are slightly different. The signals RDB* and WRB* be set up to a particular rising clock edge, and PEB* must be set up to the next falling clock edge. Regardless of which Port is involved, the 8207 will not initiate a DRAM access cycle, and may lock up, if the timing requirements are not met.

Additional logic is required to ensure that the above timing requirements are met. The signals PrtAEn* and LS2*-LS0* cannot be used directly with Port A because they become active at points too far apart in the bus cycle. These signals, however, can be OR'd with ALE to produce PEA*, PCTLA, RDA*, and WRA*. The resulting signals all become active at the same time when ALE is negated. The same reasoning applies to Port B. PEB*, RDB*, and WRB* all become active when ASQ* is asserted.

The 8207 services an access request from a Port after all preceding requests have been handled. Service

for a request from Port A or Port B begins with the 8207 enabling the address buffers for that Port. The MUX output is driven HIGH for Port A, and LOW for Port B. The logic which generates AddrAEn* and AddrBEn* ensures that both sets of address buffers will not be enabled at the same time.

When the address buffers are enabled, an address generated by the 68000 or 80186 is passed on to the shared address bus, which drives the AH7-AH0 and AL7-AL0 inputs of the 8207. The address bits present on AL7-AL0 are output on AO7-AO0 as the row address for the DRAMs. The address bits on AH7-AH0 become the column address. Since the row and column address for 64K x 1 DRAMs are only 8 bits long, AL8 and AH8 cannot be used and are tied low.

The 8207 asserts RAS* and CAS* signals at appropriate times during a DRAM access cycle. When the 8207 is programmed to interface with one bank of DRAM (as in this design), all four RAS*/CAS* pairs are activated simultaneously. In this configuration, multiple RAS*/CAS* pairs can be used to strobe the single bank of DRAMs, reducing the capacitive loading on the individual RAS* and CAS* drivers. In the emulator, one RAS*/CAS* pair strobes the low byte of the shared RAM, and another pair strobes the high byte.

The RAS*, CAS*, and AO8-AO0 outputs are designed to

directly drive the heavy capacitive loads associated with DRAM arrays. To prevent excessive ringing on these lines, series damping resistors must be provided. Intel recommends using resistors between 30 and 60 Ohm. For this design, 33 Ohm resistors on AO7-AO0 and 68 Ohm resistors on the RAS*/CAS* lines appear to be the most effective.

The 8207 also asserts the write strobe (WE) during DRAM write cycles. This signal is not designed to drive the DRAM array directly. Instead, it is NAND'd with the inverted high/low byte signals (UDS*/LDS*, BHE*/A0) from the 68000 or 80186. The high/low byte signals must be latched because the 8207 can and will change MUX before the end of a DRAM access cycle.

Separate Data In and Data Out buffers are required for each Port. The Din and Dout lines of the DRAMs in the shared RAM cannot be tied directly together because the 8207 executes "late write" cycles (CAS* is asserted before WE*). In late write cycles, there is insufficient time for the Dout outputs to be placed in a high impedance state before the Din lines are driven with write data. If the Din and Dout lines were tied together, bus contention would occur.

The output enable signals for the data buffers are generated from three signals: PSEL, PSEN, and DBM*.

The 8207 indicates the active Port by driving PSEL HIGH (for Port A) or LOW (for Port B). PSEN is asserted in conjunction with PSEL. When PSEN is HIGH, PSEL will not change state. DBM* is asserted for all read and refresh cycles. Different logic is used for the Port B Data Out buffer because the 68000 is running asynchronously with respect to the 8207.

The buffering of the lower half of the shared data bus (and, indirectly, the lower half of the 80186 data bus) to the upper half of the 68000 data bus is not a mistake. This configuration is necessary because the 68000 has a different memory interface from that of the 80186. The 68000 transfers even addressed bytes on the upper half of the data bus, while the 80186 transfers even addressed bytes on the lower half. Swapping the halves of the shared data bus is necessary so that the byte at a particular offset address in the shared RAM is the same byte for both processors.

The 8287 asserts AACK* to indicate that the bus cycle can be completed. AACK* is factored with PrtAEn* to form the ready signal (SRDY) for the 80186. Likewise, AACKB* is factored with PrtBEn* to form the ready signal (XRDY) for the Amiga.

The logic illustrated in Schematic 4 is related to the dual ported RAM array. It is used by the Amiga

console driver software (discussed in Chapter 4) to determine if the video data stored in the shared RAM has been updated by the 80186.

The 74LS175 Quad D Flipflops (U73 and U74) function together as an 8-bit tag register. Each bit in the tag register is associated with a different 8K block of address space in the upper 64K of the shared RAM. When the 80186 writes data into the one of these blocks, the corresponding bit in the tag register becomes a "1".

The Amiga console driver reads the contents of the tag register by accessing address 21F001h (in Amiga expansion space). Immediately after the read, the tag register is cleared by an access to address 21F400h. For this discussion, it is assumed that the base address of the emulator is 200000h.

NMI-Generating Logic

The logic illustrated in Schematic 3 is a solution to the the problem of I/O port accesses by IBM PC software. This logic is necessary because the Little Board/186 I/O port map is very different from the IBM PC I/O port map. Software which performs I/O operations based on the IBM PC port map will probably not run correctly on the Little Board/186.

The signal IOPA, generated by address and status decoding logic, becomes active during all 80186

accesses to the lower 1K of I/O address space. The leading (rising) edge of IOPA generates an NMI. The 74LS257 multiplexer prevents IOPA and AFR* from generating interrupt requests until the corresponding interrupt handlers have been installed. IOPA and AFR* are enabled to generate interrupts by an access to I/O address 1300h.

IOPA is also the latch enable for three 74LS373 latches. The latches store the I/O port address and status bits generated during the most recent I/O port access. The I/O data does not need to be latched because, for all I/O instructions, it is contained in the 80186 CPU registers AL or AX. The latched address and status bits can be read at I/O ports 1020h and 1030h, respectively.

Chapter 4: Theory of Operation - Software

Although it is not as evident as the hardware, software still has an important role in the emulator design. The software components of the design consist of new interrupt handlers for the Little Board/186, and a console (keyboard and video display) for the Amiga. This chapter is a discussion of these components. Commented source code can be found in Appendix B.

As in Chapter 3, the discussion in this chapter is also at a functional level. The intricacies of "C" and Intel assembly language programming are not examined. More information on these languages can be found in The C Programming Language, the iAPX 86/88, 186/188 User's Manual, Programmers Reference, The 8086 Book, and Assembly Language Primer for the IBM PC and XT.

Amiga Console Driver

The Amiga console driver handles keyboard input and video output. The driver receives raw keyboard scan codes, processes them, and then writes the processed data into the shared RAM. At periodic intervals, the driver also reads the video data stored in the shared RAM, and updates the display. The console driver software used in this design is not the same as, and should not be confused with, the "console device" I/O

handler present in the Amiga operating system.

The console driver first opens two "Intuition" Screens. "Intuition" is the window oriented, Macintosh-like user interface of the Amiga. A Screen specifies certain characteristics (number of colors, resolution, default font, etc.) for the portion of the display which it occupies. A Screen can be of any height, but it must always occupy the entire width of the display. A number of Screens can be open at any particular time, and they can partially or completely overlap each other. Two Screen are opened by the console driver to facilitate a double buffering technique for display updating.

The console driver then opens an Intuition Window in each of the Screens. All output of a process running under Intuition is done through a Window. Any number of Windows can be sent output at a particular time. Input is associated with a Window. A process running under Intuition can receive notification of input events (keypresses, mouse movements, etc.) occuring in the "active" Window. A Window is made active by program control, or by placing the mouse cursor in the Window and pressing the left mouse button. Only one Window is active for input at any particular time. More information on Intuition Screens and Windows can be found in the Amiga Intuition Manual, Inside the Amiga,

and the Amiga Programmer's Guide.

After initializing several variables, the console driver puts itself to "sleep", awaiting notification of input by the IDCMP. The IDCMP is an Intuition subtask which monitors the keyboard, mouse, Intuition clock (10 ticks/second), and disk drives. Any messages from the IDCMP will awaken the console driver, and the driver will proceed to determine the origin of the input event. It is important to note that while the console driver is "asleep", it is not using any CPU time. Other processes running concurrently with the driver can take advantage of the freed CPU time.

If the input event is a key press or key release, the console driver receives a key scan code from the IDCMP. The scan code simply indicates the relative position of the key on the keyboard. The scan code must be decoded to determine what it represents (number, character, punctuation, function key, etc.).

The console driver translates scan codes similiarly to the way in which the IBM PC ROM-BIOS translates scan codes received from the IBM PC keyboard. The ROM-BIOS organizes the keys into three general groups:

SHIFT-TOGGLE keys (left shift, right shift, caps lock, alt, ctrl, insert, num lock, scroll lock, etc.),

SPECIAL keys (function, arrow, alt-key combinations),

and ASCII keys (characters, numbers, punctuation,

ctrl-key combinations). If the scan code is determined to represent a SHIFT-TOGGLE key (by comparing the untranslated scan code against the scan codes for known keys), the ROM-BIOS updates the keyboard status bytes stored in the ROM-BIOS data area (see Mapping the IBM PC and PCjr and Programmer's Guide to the IBM PC for more details on the ROM-BIOS data area). If the scan code represents a SPECIAL key, a two byte code is entered in the key buffer (key codes are stored in the key buffer until they are needed by a program). The high byte typically contains the scan code, and the low byte contains a zero. A two byte code is also entered in the buffer for scan codes which represent ASCII keys. In this case, the low byte contains the ASCII code for the key, and the high byte contains the scan code.

The console driver first determines if the scan code represents a key press or a key release. If the scan code is less than 128, it represents a key press. A scan code greater than 127 indicates a key release.

The first scan codes that the undecoded scan code is compared against are those of the SHIFT-TOGGLE group. If a match is found, the appropriate bit in the console driver keyboard status word (kbd_status) is updated. The bit is set for a keypress, and cleared for a key release. The setting/clearing of the bits is performed by OR'ing/AND'ing kbd_status with various masks.

The left or right "Amiga" key can be pressed in conjunction with the "i" and "s" keys to emulate the insert and scroll lock toggle keys present on the IBM PC keyboard. Specific keys dedicated to these functions are not available on the Amiga keyboard. Instances of these key combinations will also cause the corresponding bits in kbd_status to be updated.

After kbd_status has been updated, it is written into the keyboard data location (kbd_dat) in the shared RAM.. A function code indicating that the contents of kbd_dat is status information is also written into the shared RAM. An 80186 interrupt request is then generated by accessing memory address 21F000h (an access to this address will pulse AFR*, which drives the INT1 interrupt request pin of the 80186). The corresponding interrupt handler (discussed later in this chapter) will copy the status data into the ROM-BIOS data area of the Little Board/186.

The next scan codes which the untranslated scan code is compared against are those of the SPECIAL group. If a match is found, the scan code is used as an index into a translation table (trans_table). Trans_table contains the two byte codes which the IBM PC ROM-BIOS would place in the key buffer in response to the same keypress. The state of bits in kbd_status determines which field in the trans_table entry should be read.

Any remaining key combination involving the alt key are also considered to be SPECIAL. These key combinations are also translated into a two byte code using trans_table.

Any scan code remaining untranslated at this point represents a key or key combination which has an ASCII value associated with it. In all cases, the scan code is used as an index into trans_table. It the ctrl status bit is set, the ctrl field of the indexed entry is read. If the ctrl bit is not set, the scan code is tested against the scan codes for the alphabetic keys. If the scan code is in this group, it it translated into a two byte code based on the shift/capslock status. Finally, scan codes which represent the numeric/punctuation keys are translated into two byte codes based on the shift status.

The translated codes resulting from all SPECIAL and ASCII keys and key combinations are written into the shared RAM in the same way. The two byte code is copied into the kbd_dat location. A function code indicating that the contents of kbd_dat is a key code is also written into the shared RAM. Finally, an AFR* interrupt request is generated to the 80186.

If the input event is an Intuition clock tick, the console driver reads the shared RAM tag register. A "1" in bit 0 of the tag register indicates that the 80186

has written data into the 8K block of shared RAM starting at B0000h (in 80186 memory space). If new data is present, the console driver clears the tag register, and proceeds to update the display. If no new data is present, the console driver simply clears the tag register, and returns to a "sleep" state.

In the current design, the console driver interprets video data as would an IBM PC Monochrome Display Adapter (MDA). The 4K of video RAM in the MDA, which is mapped into IBM PC memory space starting at address B0000h, is large enough to contain data for 2000 display characters (25 rows of 80 characters). A pair of bytes is required for each character. The high byte, called the "attribute", describes the appearance of the character (blinking, bold face, underlined, normal, etc.), and the low byte contains the ASCII code for the character.

To update the display, the console driver reads character/attribute pairs from the shared RAM starting at address 210000h (210000h in Amiga memory space maps to address B0000h in 80186 memory space). Characters with the same attribute are placed in a buffer until the end of the row (80 characters) or a character with a different attribute is reached. All characters in the buffer are then output to the Window (back_window) in the Screen (back_screen) which is currently behind the

Screen (front_screen) which is being displayed. Characters with the new attribute, or in the new row, are read into the buffer until a different attribute or the end of the row is reached. Again, the characters in the buffer are output to back_window. This process continues until all 25 rows have been output. Back_screen and front_screen are then switched. The double buffering of the Screens hides the row by row update process, making a new view appear almost instantaneously.

The console driver interprets the attribute byte slightly differently from the MDA. Certain characteristics, such as blinking, could not be easily emulated on the Amiga display. Therefore, new meanings have been assigned to certain attributes. The mapping of the MDA interpretation to the console driver interpretation is illustrated in Figure 4.1.

	MDA	Amiga display
00h	non-display	non-display
01h	underlined	current attribute
07h	normal	white on blue
70h	reverse video	blue on white
08h	non-display	non-display
09h	highlighted underlined	orange on blue
0Fh	highlighted normal	orange on blue
78h	highlighted reverse video	blue on orange
80h	non-display	non-display
81h	blinking underlined	black on blue
87h	blinking normal	black on blue
F0h	blinking reverse video	blue on black
88h	non-display	non-display
89h	blinking highlighted underlined	black on blue
8Fh	blinking highlighted normal	black on blue
F8h	blinking highlighted reverse videeo	blue on black

only these attribute bytes are meaningful to the MDA

MDA to Amiga Display Attribute Map

Figure 4.1

Little Board/186 Interrupt Handlers

The interrupt handlers found in the ROM-BIOS of the IBM PC and PC-compatibles provide a standard way of transferring data to and from system hardware. The calling conventions and the format of return values are the same for all PC-compatibles. The operations performed within the handlers, however, are tailored around the hardware in the particular system.

The ROM-BIOS interrupt handlers can be invoked in one of two ways. Some handlers are executed in response to a hardware event. For example, the NMI handler is executed when a low to high transition occurs on the NMI pin of the CPU. Another example would be the keyboard hardware interrupt handler. The arrival of a key scan code at the keyboard port generates a interrupt request. In response, the handler services the request by reading the scan code form the keyboard port, processing the scan code, and then writing the translated data into the key buffer.

The second way in which ROM-BIOS interrupt handlers can be invoked is via the Intel assembly language "INT" instruction. Applications programs typically call handlers in this manner when it necessary to access system hardware or data areas managed by other ROM-BIOS routines. Services provided by ROM-BIOS interrupt

handlers include the reading and writing of disk sectors, the retrieval of key codes from the key buffer, and the manipulation of the video display. The arguments for these services and any resulting return values are usually passed in CPU registers.

The Little Board/186 requires new interrupt handlers to allow IBM PC application software to access emulator hardware in a standard way. The AMPRO ROM-BIOS keyboard hardware handler reads keyboard data from the 2761 serial controller. The AMPRO ROM-BIOS video interrupt handler writes data to the 2761. The new interrupt handlers for these functions read data from and write to the shared RAM.

The new NMI handler is initiated as a result of an I/O port access by an IBM PC application program. The handler begins execution immediately after the I/O instruction which caused the NMI. The address and status bits present in the latches of the NMI-generating logic when the NMI handler begins execution are those generated by the I/O instruction.

The current version of the new NMI handler simply reads the latched address and status bits, converts them to ASCII strings, and then writes the strings to the display. This handler also writes a message indicating that an I/O port access has occurred. The NMI handler then terminates the application program by calling the

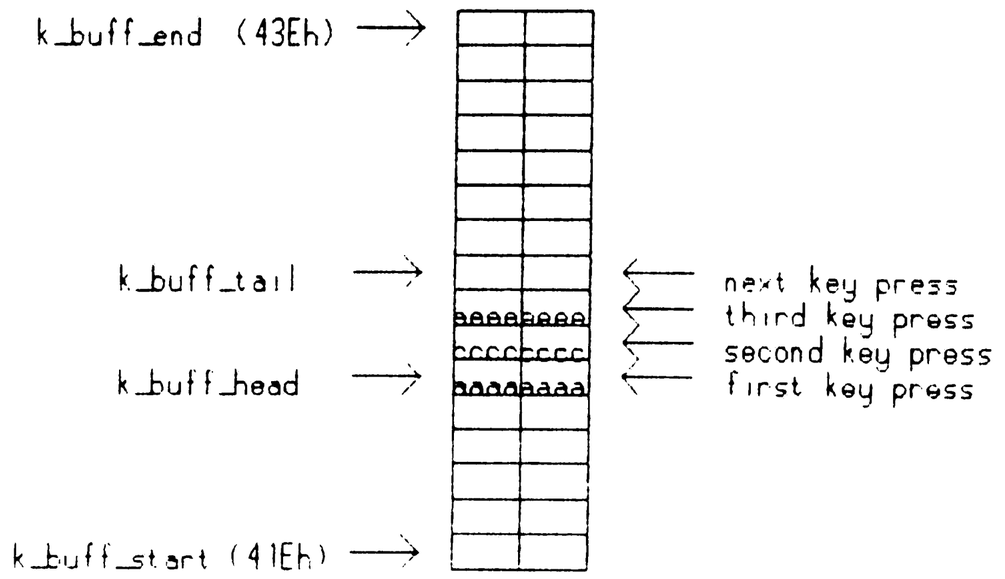
PC-DOS/MS-DOS program terminate function (INT 21h, AX = 4C00h). More information on program termination can be found in Advanced MSDOS and Programmer's Guide to the IBM PC.

The new keyboard hardware interrupt handler is invoked in response to an interrupt request on INT1. The Amiga console driver generates an interrupt request on INT1 to inform the 80186 that new keyboard data is present in the shared RAM.

The keyboard hardware handler first determines (by examining the contents of afr_dat) if the keyboard data is a two byte key code or a status word. If the data is a status word, it is copied into the keyboard flag bytes locations (417h, 418h) in the ROM-BIOS data area. If the data is a key code, it is entered in the key buffer, also stored in the ROM-BIOS data area.

The key buffer has room for 15 two byte key codes. Address pointers (stored in k_buff_head and k_buff_tail) indicate the head and tail of the buffer (see Figure 4.2). When the buffer is empty, the head and tail pointers point to the same key code location in the key buffer. When a key code is read from the shared RAM, the code is entered in the key buffer at the location pointed to by the tail pointer. The tail pointer is then incremented by two to point to the next key code location. When the tail pointer reaches the

end of the buffer, it is reset to point to the start of the buffer.



Key Buffer

Figure 4.2

The new keyboard I/O interrupt handler provides three services which can be called by application programs to obtain keyboard data. The "read character" service (INT 16h, AH = 0) reads a key code from the key buffer, and returns it register AX. The service reads the key code pointed to by the key buffer head pointer, and then sets the head pointer to point to the next key code location in the buffer. If the buffer is empty, the service waits until a key code arrives. The "report character" service (INT 16h, AH = 1) simply determines if the key buffer is empty, and returns without waiting. The "get shift status" service (INT 16h, AH = 2) returns the keyboard flag byte stored in address 417h in the ROM-BIOS data area.

The new video interrupt handler provides only one service, the "write tty" service (INT 10h, AH = 0Eh).

The "write tty" service handles character output to the video display. This service formats the character output to make the display appear like a tty terminal. Character data appears on the display at the current cursor position, and then the cursor is moved right one column. When the cursor reaches the end of a row, it is moved to the first column of the row below. When the display becomes full, the rows of characters are scrolled up, and a new row begins at the bottom of the display.

When the "write tty" service begins execution by testing the character argument to see if it is a backspace, linefeed, or carriage return. If argument is one of these characters, the cursor position (row number / column number) stored in the ROM-BIOS data area is altered appropriately. The new cursor position is also written into the shared RAM. The Amiga console driver reads the cursor position, and moves the cursor on the display.

The "write tty" service writes all other characters which it receives as arguments into the video RAM portion of the shared RAM. Each row/column position on the display corresponds to a particular offset word address in video RAM. The word address is calculated using the equation: address = ((160 * row number) + (2 * column number)). The character is assigned a

"normal" attribute, and the character/attribute pair is written into video RAM at the calculated offset address.

When the display becomes full, it is necessary to scroll the rows of characters up, and start a new line at the bottom of the display. This is accomplished by moving all the video data except the data making up the top row down (toward low memory) a distance of 160 bytes (the top row of the display is at the bottom of video RAM). The data that made up the top row will be written over by the data of the second row, the data of the previous second row will be written over by data from the third row, and so on. The character bytes in the bottom row are filled with "space" characters to form a blank line on the display.

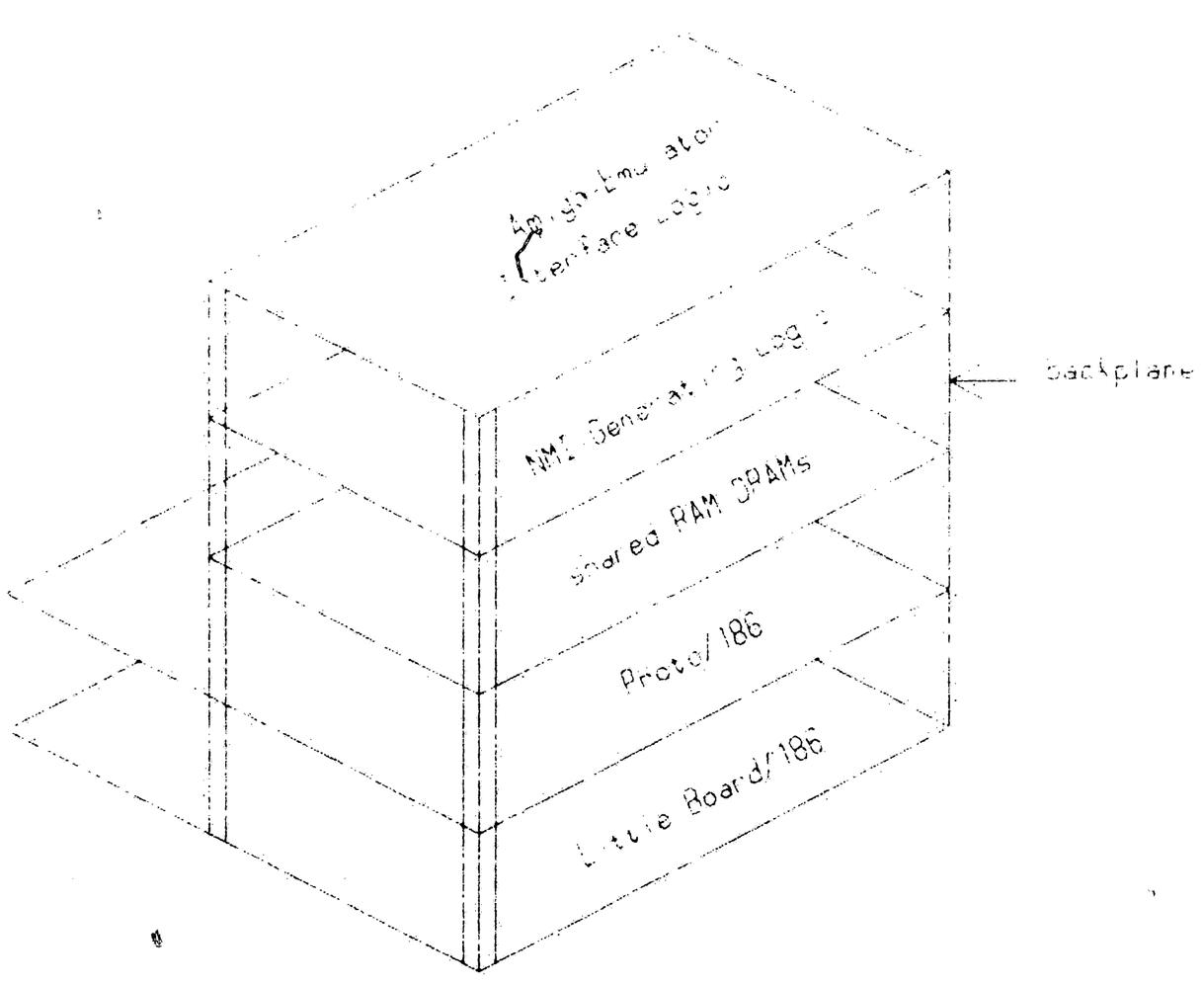
The new interrupt handlers are implemented as procedures in a *terminate and stay resident* (TSR) program. A TSR program executes similiarly to other disk-based IBM PC programs, except that when it finishes execution, it remains in memory. The memory which was allocated for the program is not freed for reallocation to other programs. Any code in a TSR program remains in memory, and is not written over by other programs.

The "main" code of the TSR implemented for the emulator design sets interrupt vectors 2, 14, 22, and 16 to point to the new NMI handler, keyboard hardware handler, keyboard I/O handler, and video handler,

respectively (see the iAPX 86/88, 186/188 User's Manuals, The 8086 Book, and Programmer's Guide to the IBM PC for more information on interrupt vectors). The "main" code also initializes the ROM-BIOS data area with values compatible with the new interrupt handlers. The NMI-generating logic and INT1 are enabled. The program then terminates using the terminate and stay resident service (INT 21h, AX = 3100h).

Chapter 5: Construction of Prototype

A wirewrapped prototype of the emulator design was constructed using three Vector boards, the Proto/186, and the Little Board/186 as illustrated in Figure 5.1. The 8207, along with its support logic, is mounted on the Proto/186. The DRAMs making up the shared RAM are mounted on the bottom Vector board. The next higher Vector board contains the NMI-generating logic, and the highest board contains the Amiga-emulator interface logic. A backplane connects the Vector boards to each other, and to the Proto/186.



Prototype of Emulator
Figure 5.1

A grid system supplies power to the chips on the Proto/186. Power and ground wires are laid both

horizontally and vertically across the board. The power and ground pins of the chips are wired to the intersections in the grid. A similiar layout of power and ground lines is employed on the highest two Vector boards. Grids are not required on the DRAM Vector board because power and ground planes are built into this board.

The principal communication link between the Vector boards in the previously mentioned backplane. The buffered 68000 address bus, the DRAM address and control signals generated by the 8207, and various other signals are conducted via the backplane. The buffered 68000 and 80186 data busses, however, are conducted from board to board on ribbon cable. Ribbon cable also connects the Amiga-emulator interface logic to the 86-contact card edge connector which plugs onto the Amiga expansion bus.

Each TTL IC is bypassed by one .1 microfarad ceramic disk capacitor. The capacitors, connected between the power and ground pins of the ICs, dampen the current spike which occurs when a TTL IC changes state. The 4164 DRAMs also require bypass capacitors (three .1 microfarad capacitors per chip) because they generate current spikes when the RAS* and CAS* lines change state.

Chapter 6: Evaluation of Design

The prototype of the emulator successfully booted PC-DOS, and ran several IBM PC programs. Using DEBUG, it was possible to modify the video display by writing data directly into the video RAM portion of the shared RAM. Therefore, other programs which directly access video RAM should also run properly on the emulator.

Although the current emulator design can run some IBM PC software, its usefulness would be increased significantly by certain hardware and software modifications. As mentioned in Chapter 3, there is a tendency for loss of data in the DRAM on the Little Board/186. This problem could be solved by reprogramming DMA channel 1 to refresh only the lower 512K of 80186 memory space, rather then the entire 1 megabyte. The DMA channel would then not have to access the shared RAM, and contend with accesses by the 68000.

Another hardware improvement would be to replace the 8207/buffer/DRAM logic with dual ported static RAM chips. This change would reduce the number of chips and the amount of board space required for the shared RAM. Dual ported static RAMs were not used in the current design because these chips are rather difficult to obtain in small quantities.

The Amiga console driver could be improved by

slightly altering the video refresh algorithm. Instead of redrawing the entire display every time it detects that new video data is present, the console driver could compare the video data in the shared RAM against a copy of the video data which was used for the previous display refresh. The console driver would determine which bytes had been changed, and then redraw only those portions of the display. This would considerably reduce the video refresh time required for small changes in the display.

The amount of IBM PC software which the emulator could run would be significantly increased by providing more services in the video interrupt handler. IBM PC ROM-BIOS provides services for changing the cursor position, reading and writing a character at the cursor position, scrolling, etc. Many IBM PC programs use these services, as well as the write tty service implemented in the current emulator design.

Finally, it may be possible to write software which would allow the emulator to store programs and data on Amiga disk drives. Most of the hardware needed to support this operation is already present in the current emulator design.

Chapter 7: Conclusion

This thesis has presented the design, construction, and evaluation of an IBM PC emulator for the Amiga 1000. Most of the "what" and "how" questions have been answered. Reasonable questions to ask at this point are the "who" and "why" questions, such as "Who would use the emulator?" and "Why would someone use the emulator instead of an actual IBM PC?".

Certain people would definitely find the emulator to be of little use. If a person requires the use of IBM PC hardware expansion cards, he would be better off with an actual IBM PC. The emulator discussed in this thesis, like most emulators, does not have hardware expansion slots. A person who wants to run programs which require 100% hardware compatibility with the IBM PC should also avoid the emulator. The emulator will never be 100% hardware compatible with the IBM PC.

People who would find the emulator useful are those who already have an Amiga 1000, but still want to run, on occasion, fairly "well behaved" software written for the IBM PC. "Well behaved" programs do not require the presence of particular hardware (other than the CPU) at a particular address in order to run correctly. Many IBM-PC productivity (business, CAD, word processing) programs are fairly well behaved, requiring only an

equivalent for video RAM. If the modifications mentioned in Chapter 6 are incorporated in the design, the emulator should run many, if not all, well behaved programs.

If a person wants to run IBM PC software, however, why not simply buy an IBM PC or an IBM PC compatible computer? The emulator offers significantly more computing power than a typical 4.77 Mhz IBM PC compatible computer. The cost of the emulator, however, is only slightly more than a complete IBM PC compatible system (including a keyboard, monitor, disk drive, and any necessary expansion cards).

In conclusion, the practicality of the emulator discussed in this thesis varies depending on the needs of particular users. In its current state, the emulator is only marginally useful. If the modifications mentioned in Chapter 6 are incorporated in the design, however, the Amiga-emulator system could run nearly all well behaved IBM PC software.

Bibliography

Berry, John Thomas. Inside the Amiga.
Indianapolis, IN: Howard W. Sams & Co., 1986.

Crockett, Larry. "MacCharlie".
BYTE. February 1986. p. 262.

Davies, Russ. Mapping the IBM PC and PCjr.
Greensboro, NC: Compute! Publications, 1985.

Designing Hardware for the Amiga Expansion Architecture.
West Chester, PA: Commodore-Amiga, Inc., 1986.

Duncan, Ray. Advanced MSDOS.
Redmond, WA: Microsoft Press, 1986.

Hofacker, Winfred. Microcomputer Hardware Handbook.
Pomona, CA: ELCOMP Publishing, Inc., 1982.

iAPX 86/88, 186/188 User's Manual, Hardware Reference.
Santa Clara, CA: Intel Corporation, 1985.

iAPX 86/88, 186/188 User's Manual, Programmer's Reference. Santa Clara, CA: Intel Corporation, 1985.

Kane, Gerry. 68000 Microprocessor Handbook.
Berkeley, CA: OSBORNE/McGraw-Hill, 1981.

Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Lafore, Robert. Assembly Language Primer for the IBM PC & XT. New York, NY: The Waite Group, 1984.

Lancaster, Don. TTL Cookbook.
Indianapolis, IN: Howard W. Sams & Co., 1974.

Little Board/186 Technical Manual.
Sunnyvale, CA: AMPRO Computers, Inc., 1985.

Memory Component Handbook.
Santa Clara, CA: Intel Corporation, 1987.

Mical, R.J., and Deyl, S. Amiga Intuition Manual.
West Chester, PA: Commodore-Amiga, Inc., 1985.

Microprocessor and Peripheral Handbook, Volumes 1 and 2.

Santa Clara, CA: Intel Corporation, 1987.

Model 2AP Prototype Adapter Technical Manual. Sunnyvale, CA: AMPRO Computers, Inc., 1985.

Mortimore, Eugene P. Amiga Programmer's Guide, Volume 1. Berkeley, CA: SYBEX, 1987.

Norton, Peter. Inside the IBM-PC, Revised and Enlarged. New York, NY: Prentice Hall Press, 1986.

Norton, Peter. Programmer's Guide to the IBM PC. Redmond, WA: Microsoft Press, ????.

Peck, R., Deyl, S., and Miner, J. Amiga Hardware Manual. West Chester, PA: Commodore-Amiga, Inc., 1985.

Peck, R., Sassenthral, C., and Deyl, S. Amiga ROM Kernel Manual, Volumes 1 and 2. West Chester, PA: Commodore-Amiga, Inc., 1985.

Rector, Russell and Alexy, George. The 8086 Book. Berkeley, CA: OSBOURNE/McGraw-Hill, 1980.

Sargent III, Murry and Shoemaker, Richard L. The IBM PC from the Inside Out, Revised Edition. Reading, MA: Addison-Wesley Publishing Company Inc., 1986.

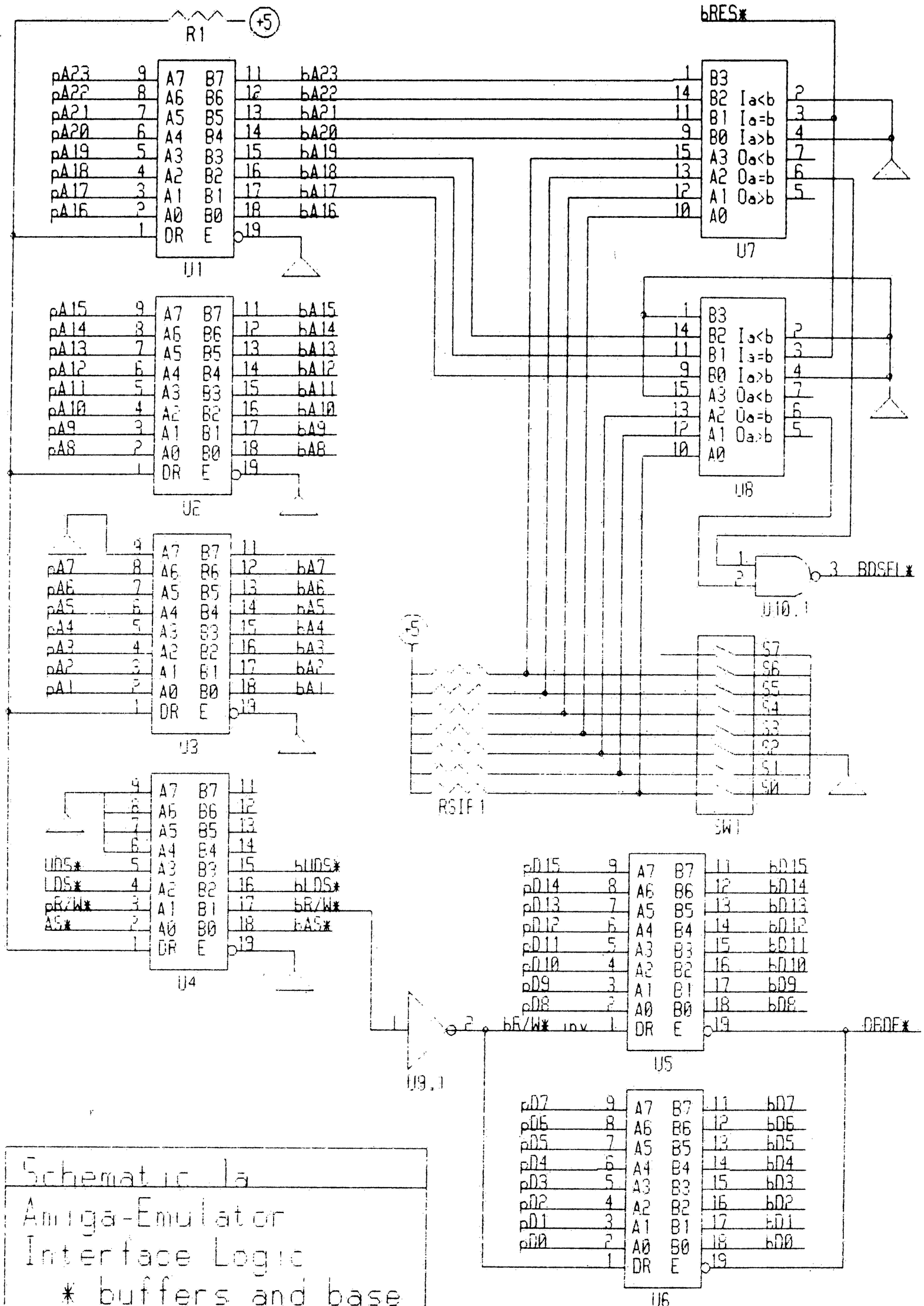
Scanlon, Leo J. The 68000: Principles and Programming. Indianapolis, IN: Howard W. Sams & Co., Inc, 1981.

Voelcker, John. "Making your PC behave like another". IEEE Spectrum. October 1986. p. 61.

Wallace, Louis R. "Bridge Over Troubled Waters". AmigaWorld. February 1988. p. 20.

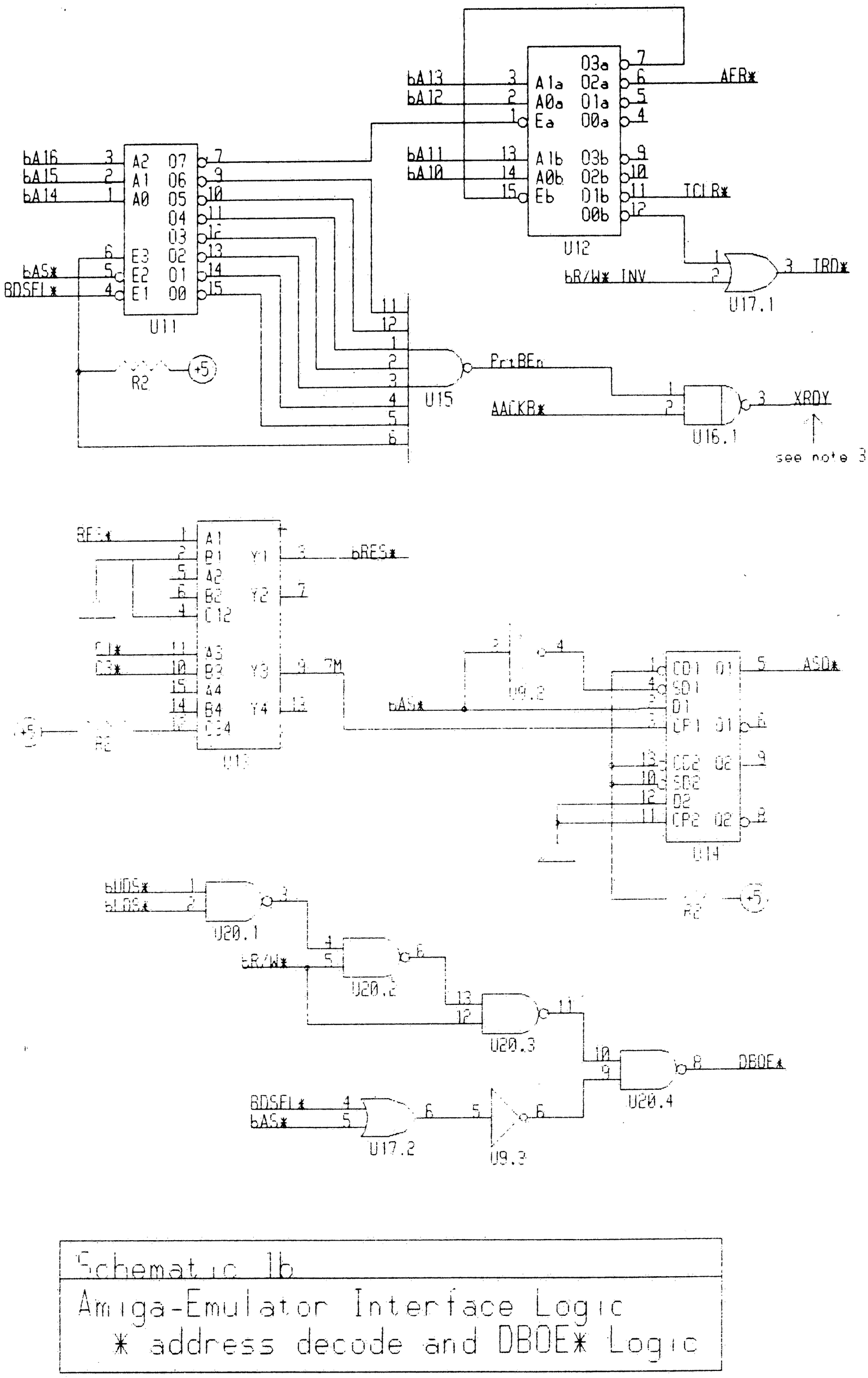
Appendix A

Schematics



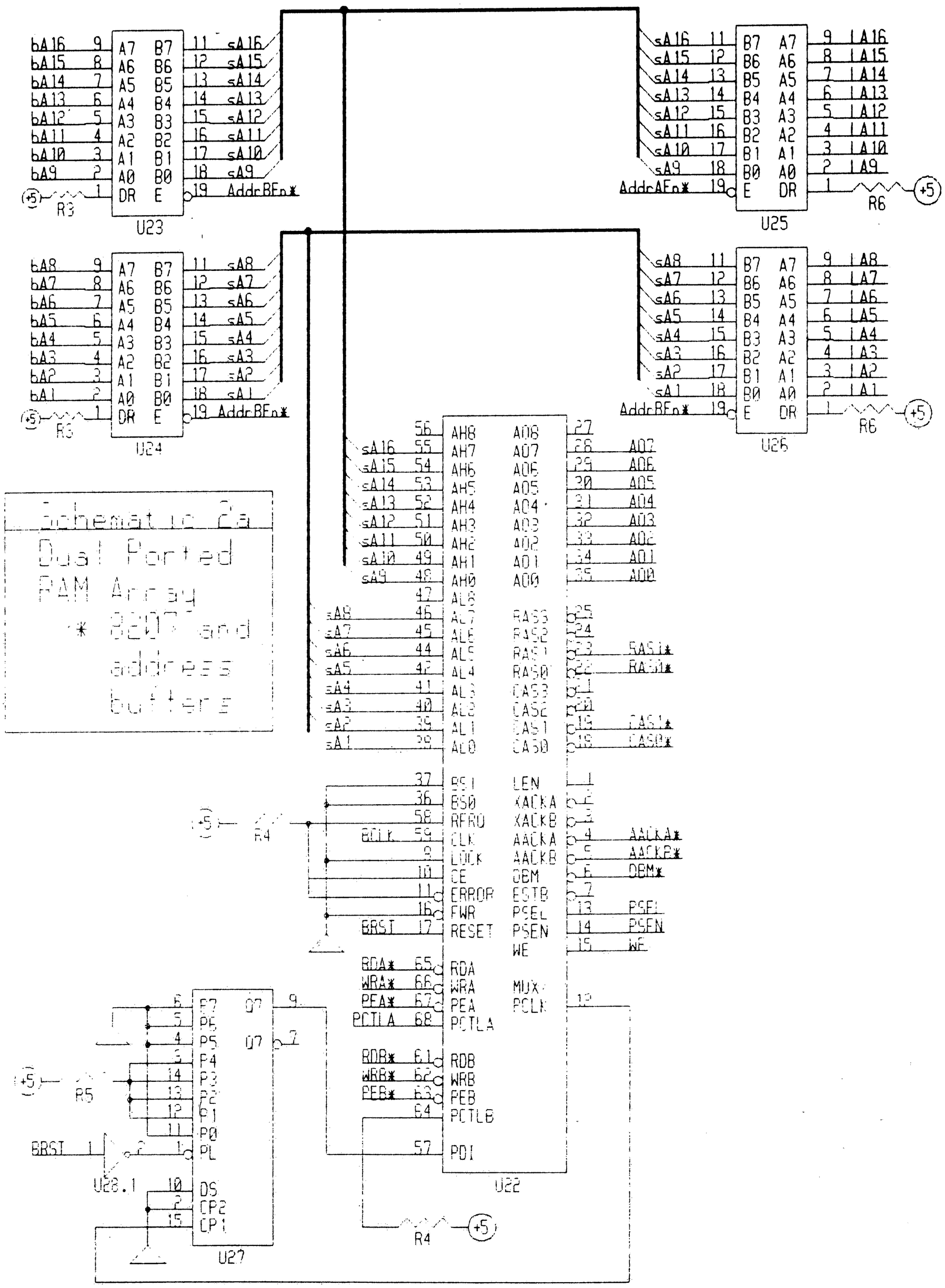
Schematic 1a

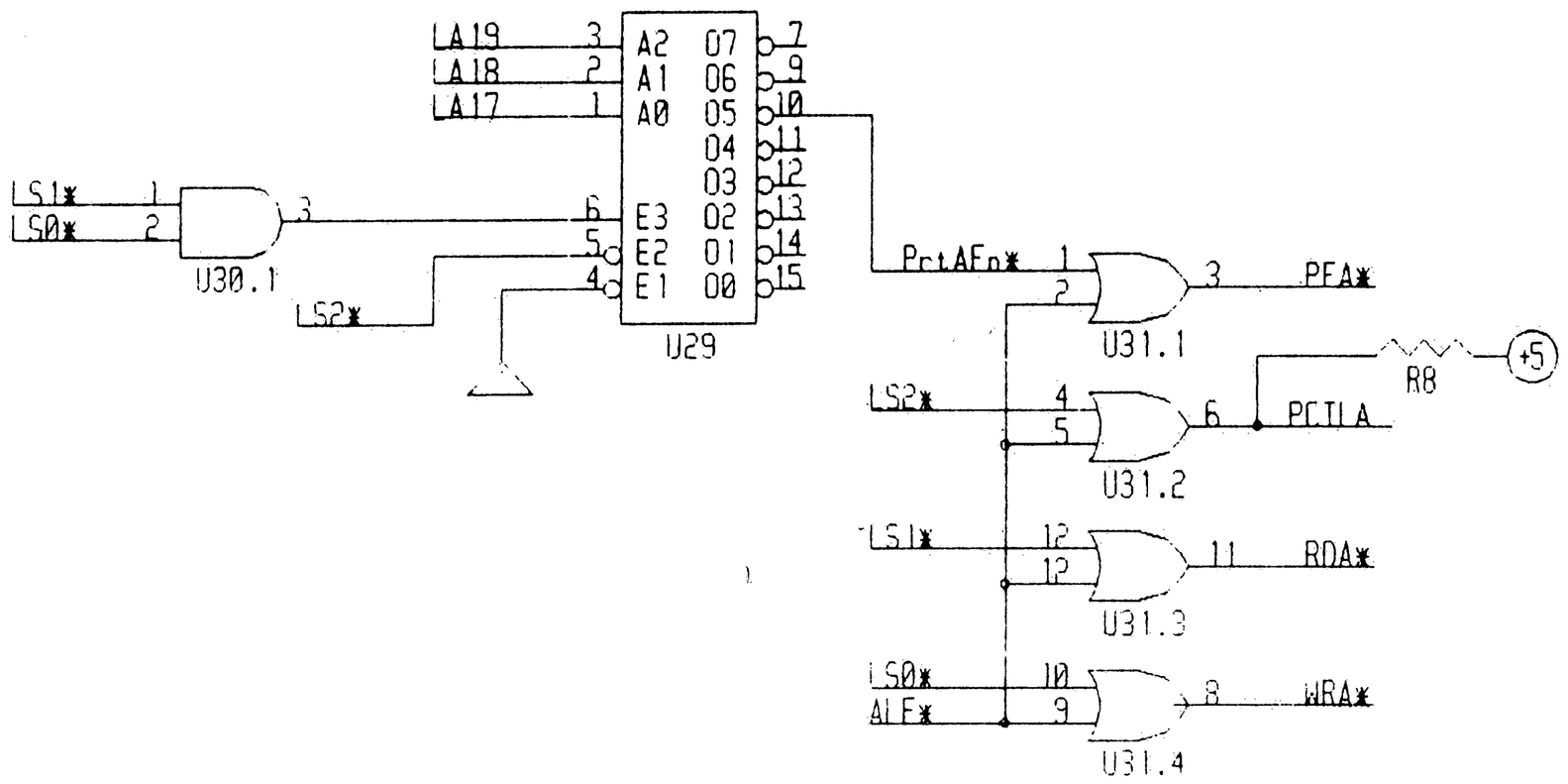
Amiga-Emulator
Interface Logic
* buffers and base
address switches

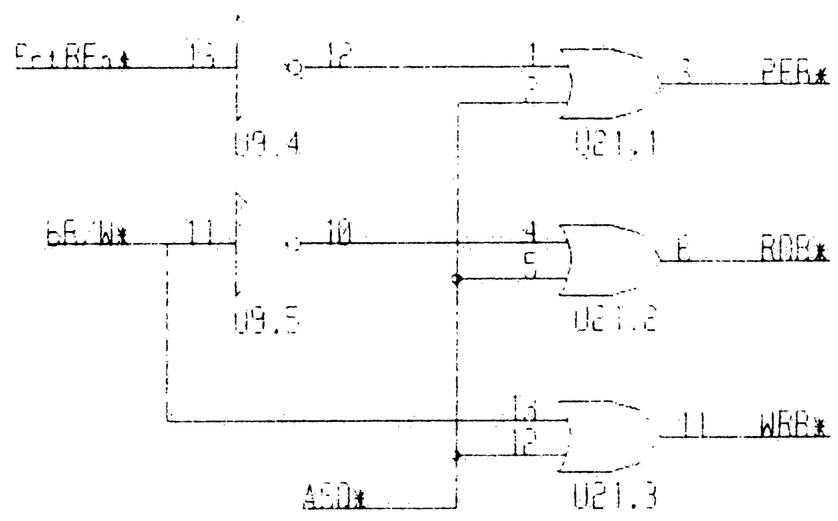


Schematic 1b

Amiga-Emulator Interface Logic
* address decode and DBOE* Logic


bA16 9
bA15 8
bA14 7
bA13 6
bA12 5
bA11 4
bA10 3
bA9 2
A7 B7
A6 B6
A5 B5
A4 B4
A3 B3
A2 B2
A1 B1
A0 B0
DR E
1
11 sA16
12 sA15
13 sA14
14 sA13
15 sA12
16 sA11
17 sA10
18 sA9
19 AddrBEn*
+5
R3
U23
sA16 11
sA15 12
sA14 13
sA13 14
sA12 15
sA11 16
sA10 17
sA9 18
AddrAEn* 19
9 lA16
8 lA15
7 lA14
6 lA13
5 lA12
4 lA11
3 lA10
2 lA9
R6
U25
bA8 9
bA7 8
bA6 7
bA5 6
bA4 5
bA3 4
bA2 3
bA1 2
11 sA8
12 sA7
13 sA6
14 sA5
15 sA4
16 sA3
17 sA2
18 sA1
19 AddrBEn*
R3
U24
sA8 11
sA7 12
sA6 13
sA5 14
sA4 15
sA3 16
sA2 17
sA1 18
AddrBEn* 19
9 lA8
8 lA7
7 lA6
6 lA5
5 lA4
4 lA3
3 lA2
2 lA1
R6
U26
Schematic 2a
Dual Ported
RAM Array
* BRST and
address
buffers
56 AH8 AD8 27
sA16 55 AH7 AD7 28 AD7
sA15 54 AH6 AD6 29 AD6
sA14 53 AH5 AD5 30 AD5
sA13 52 AH4 AD4 31 AD4
sA12 51 AH3 AD3 32 AD3
sA11 50 AH2 AD2 33 AD2
sA10 49 AH1 AD1 34 AD1
sA9 48 AH0 AD0 35 AD0
47 AL8
sA8 46 AL7 RAS3 25
sA7 45 AL6 RAS2 24
sA6 44 AL5 RAS1 23 RAS1*
sA5 43 AL4 RAS0 22 RAS1*
sA4 41 AL3 CAS3 1
sA3 40 AL2 CAS2 20
sA2 39 AL1 CAS1 19 CAS1*
sA1 38 AL0 CAS0 18 CAS0*
37 BS1 LEN 1
36 BS0 XACKA 2
58 RFRQ XACKB 3
BCLK 59 CLK AACKA 4 AACKA*
9 LOCK AACKB 5 AACKB*
10 CE OBM 6 OBM*
11 ERROR ESTB 7
16 FWR PSEL 13 PSEL
BRST 17 RESET PSEN 14 PSEN
WE 15 WE
RDA* 65 RDA
WRA* 66 WRA MUX*
PEA* 67 PEA PCLK 19
PCTLA 68 PCTLA
RDB* 61 RDB
WRB* 62 WRB
PEB* 63 PEB
64 PCTLB
57 PDI
U22
+5
R4
+5
R5
6 P7 Q7 9
5 P6
4 P5 Q7 7
3 P4
14 P3
13 P2
12 P1
11 P0
BRST 1
2
1 PL
U28.1
10 DS
2 CP2
15 CP1
U27

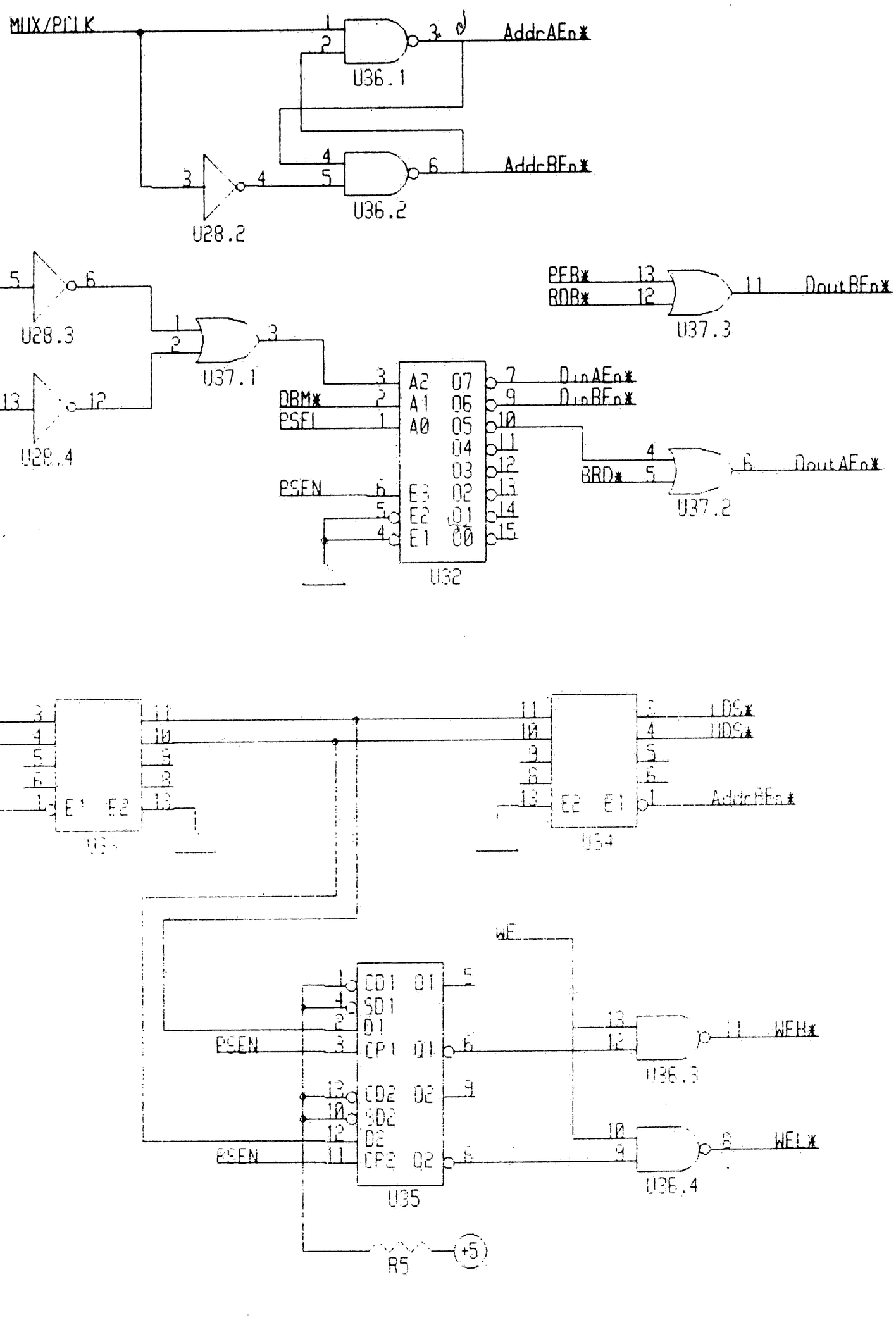




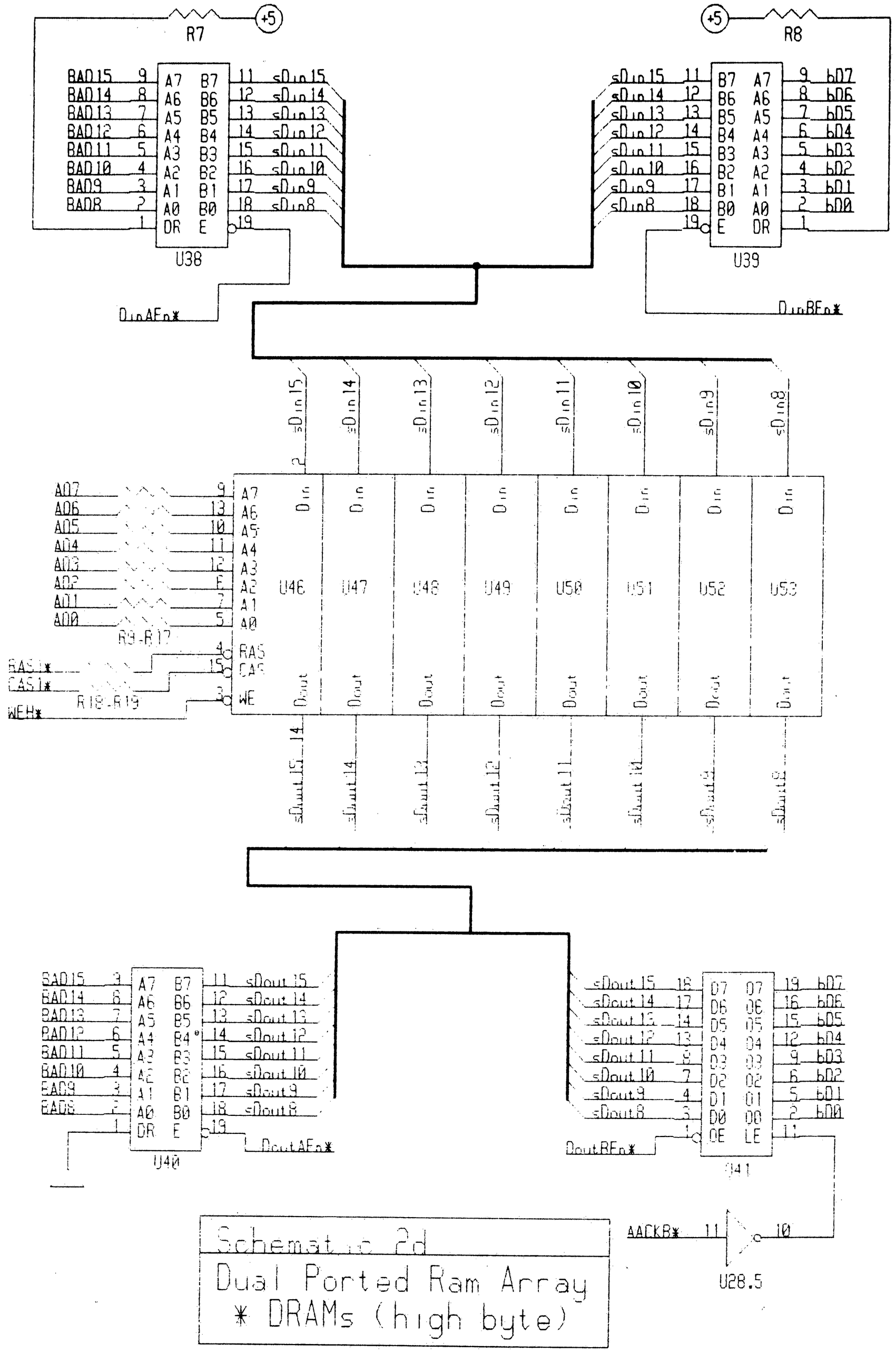
Schematic Pb

Dual Ported Ram Array

* port enable logic

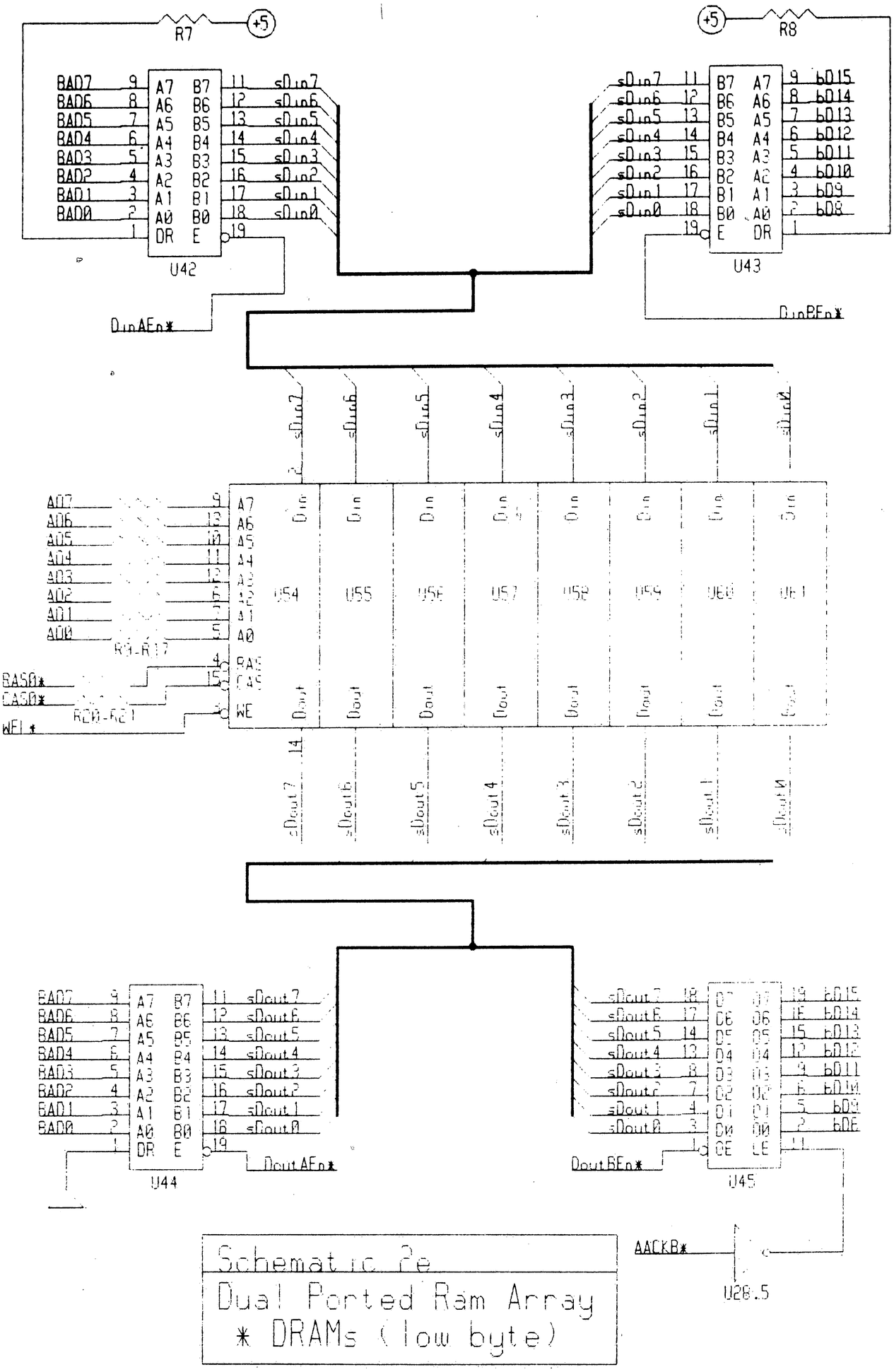


Schematic 2c
Dual Ported Ram Array * support logic



Schematic 2d

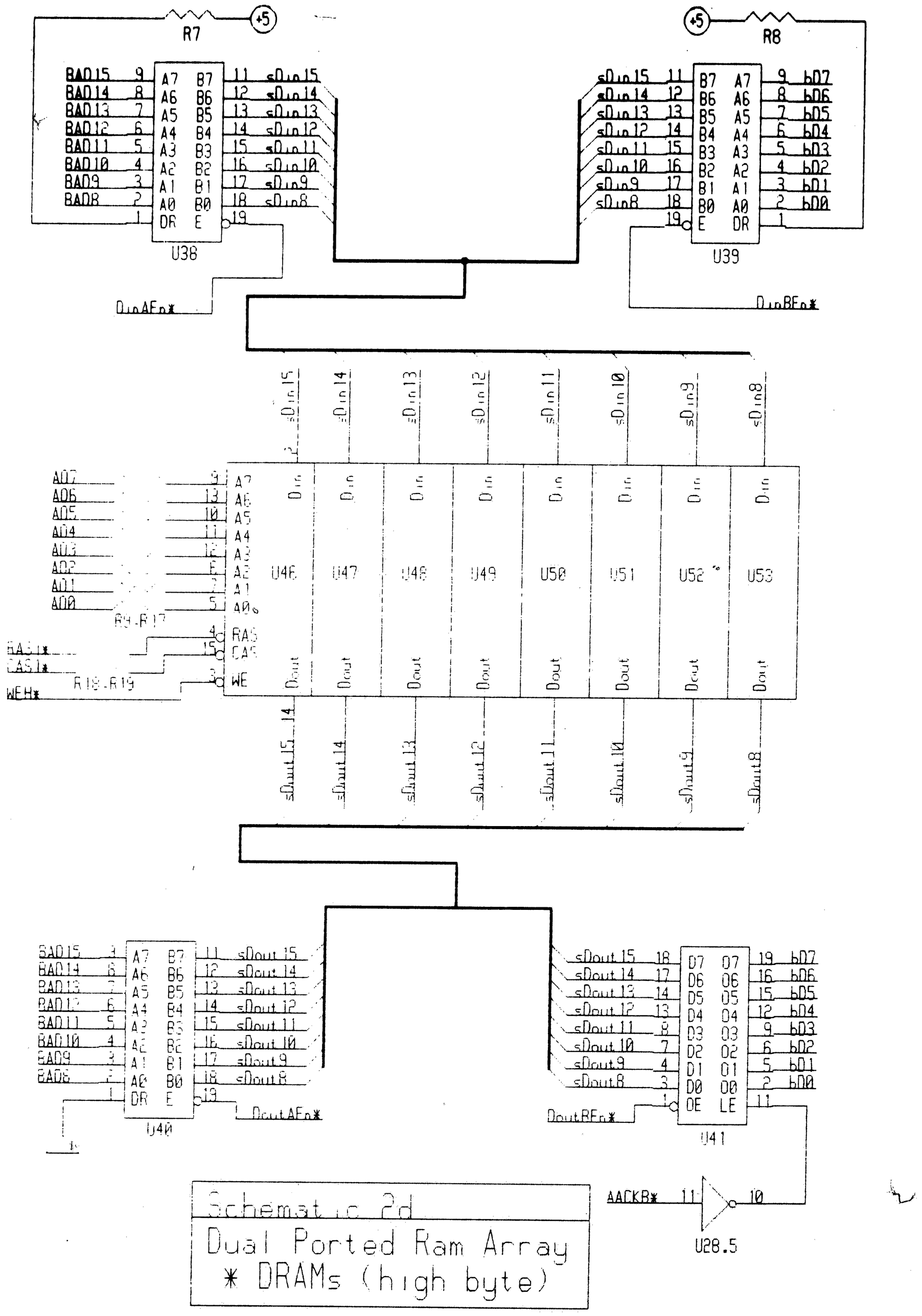
Dual Ported Ram Array
* DRAMs (high byte)

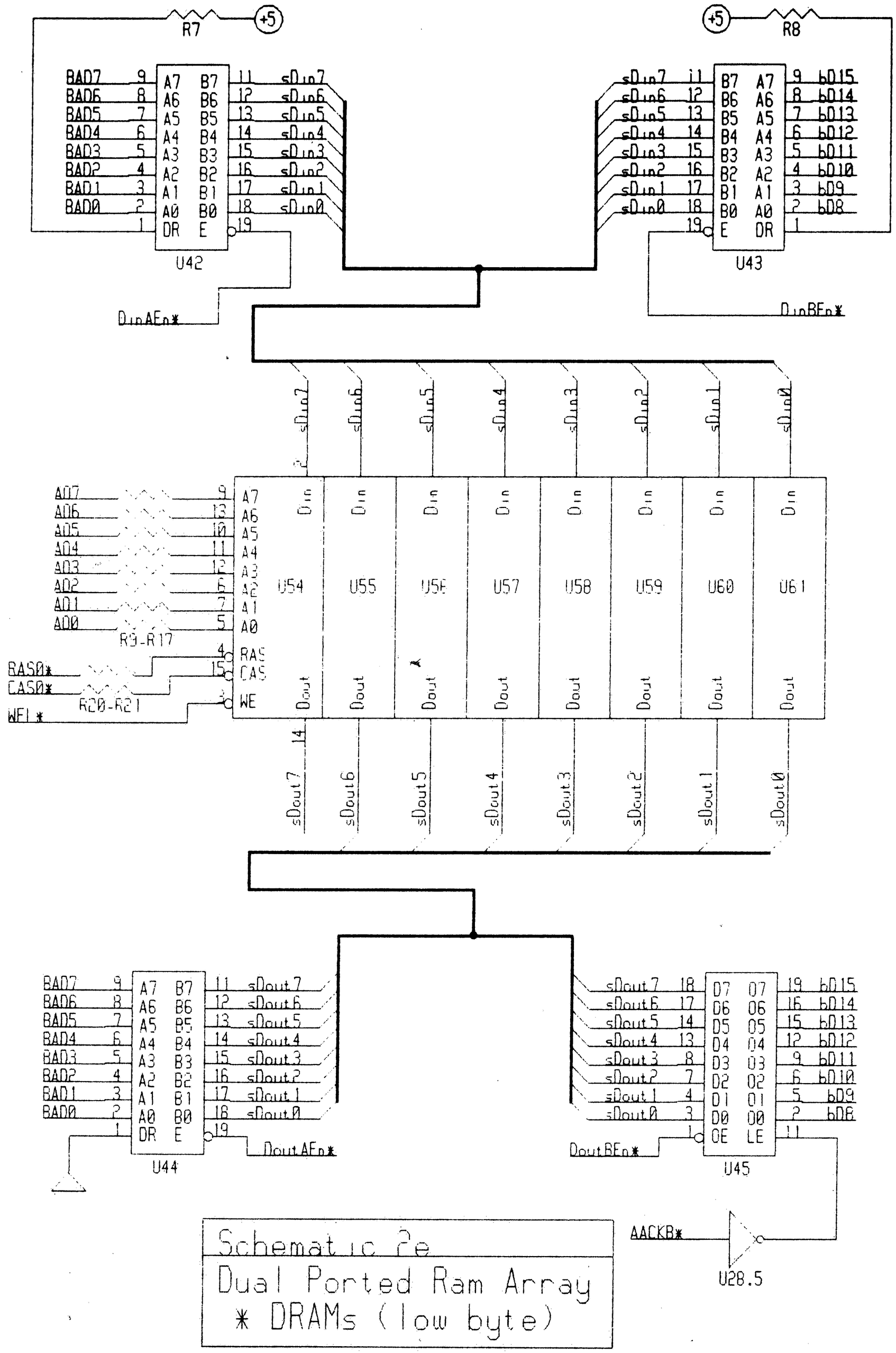


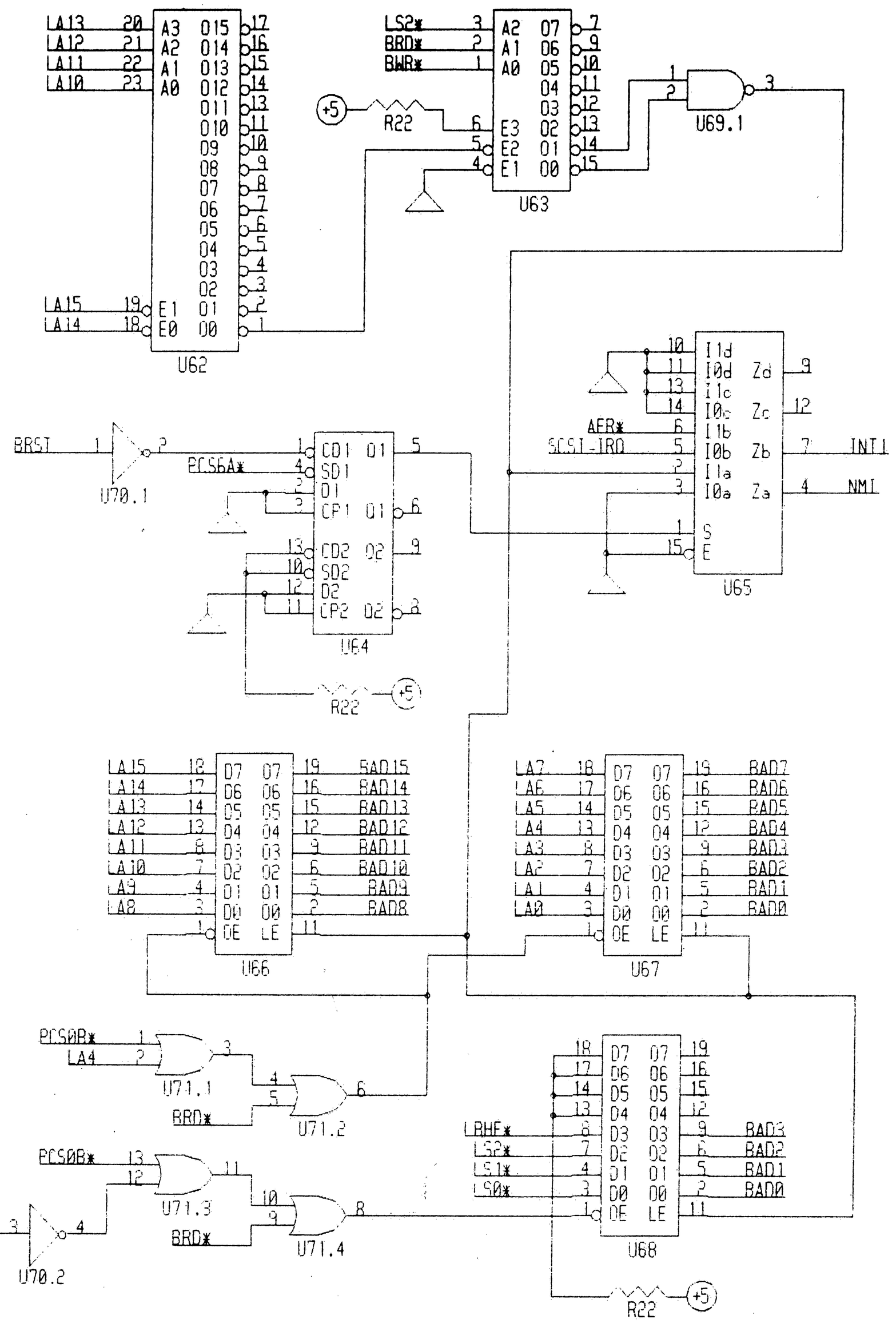
Schematic Pe

Dual Ported Ram Array

* DRAMs (low byte)

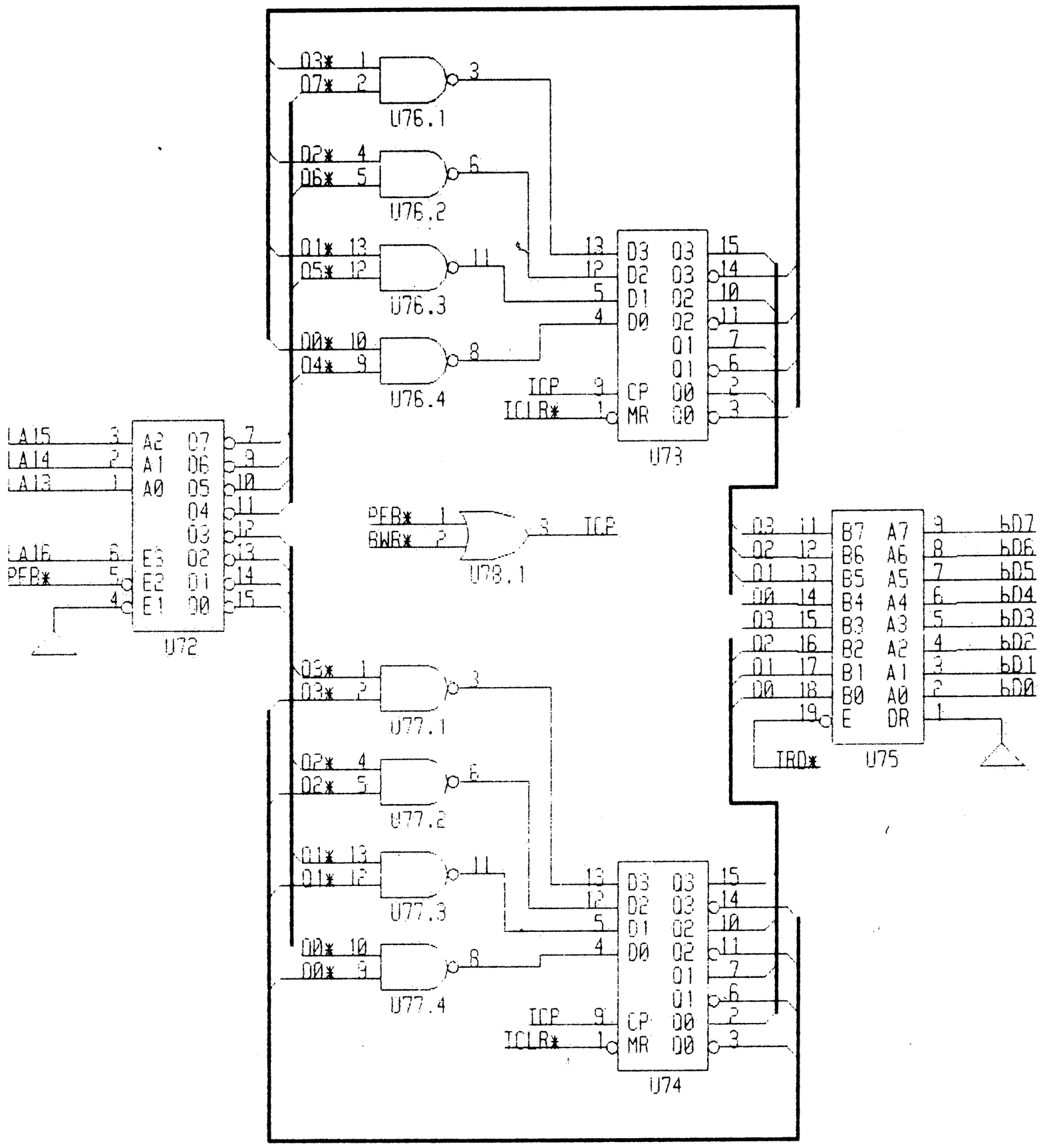

R7
+5
R8
+5
BAD15 9
BAD14 8
BAD13 7
BAD12 6
BAD11 5
BAD10 4
BAD9 3
BAD8 2
1
A7 B7
A6 B6
A5 B5
A4 B4
A3 B3
A2 B2
A1 B1
A0 B0
DR E
11 sDout15
12 sDout14
13 sDout13
14 sDout12
15 sDout11
16 sDout10
17 sDout9
18 sDout8
19
U38
sDin15 11
sDin14 12
sDin13 13
sDin12 14
sDin11 15
sDin10 16
sDin9 17
sDin8 18
19
B7 A7
B6 A6
B5 A5
B4 A4
B3 A3
B2 A2
B1 A1
B0 A0
E DR
9 bD7
8 bD6
7 bD5
6 bD4
5 bD3
4 bD2
3 bD1
2 bD0
1
U39
DinAEn*
DinBEn*
sDin15
sDin14
sDin13
sDin12
sDin11
sDin10
sDin9
sDin8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
9
13
10
11
12
15
14
5
A7
A6
A5
A4
A3
A2
A1
A0
Din
U46
U47
U48
U49
U50
U51
U52
U53
Dout
RAS
CAS
WE
4
15
3
RAS1*
CAS1*
WEH*
R9-R17
R18-R19
sDout15
sDout14
sDout13
sDout12
sDout11
sDout10
sDout9
sDout8
14
BAD15 3
BAD14 6
BAD13 7
BAD12 6
BAD11 5
BAD10 4
BAD9 3
BAD8 2
1
A7 B7
A6 B6
A5 B5
A4 B4
A3 B3
A2 B2
A1 B1
A0 B0
DR E
11 sDout15
12 sDout14
13 sDout13
14 sDout12
15 sDout11
16 sDout10
17 sDout9
18 sDout8
19
DoutAEn*
U40
sDout15 18
sDout14 17
sDout13 14
sDout12 13
sDout11 8
sDout10 7
sDout9 4
sDout8 3
1
D7 O7
D6 O6
D5 O5
D4 O4
D3 O3
D2 O2
D1 O1
D0 O0
OE LE
19 bD7
16 bD6
15 bD5
12 bD4
9 bD3
6 bD2
5 bD1
2 bD0
11
DoutBEn*
U41
AACKB* 11
10
U28.5
Schematic 2d
Dual Ported Ram Array
* DRAMs (high byte)


R7
+5
R8
+5
BAD7 9
BAD6 8
BAD5 7
BAD4 6
BAD3 5
BAD2 4
BAD1 3
BAD0 2
1
A7 B7
A6 B6
A5 B5
A4 B4
A3 B3
A2 B2
A1 B1
A0 B0
DR E
11 sDin7
12 sDin6
13 sDin5
14 sDin4
15 sDin3
16 sDin2
17 sDin1
18 sDin0
19
U42
sDin7 11
sDin6 12
sDin5 13
sDin4 14
sDin3 15
sDin2 16
sDin1 17
sDin0 18
19
B7 A7
B6 A6
B5 A5
B4 A4
B3 A3
B2 A2
B1 A1
B0 A0
E DR
9 bD15
8 bD14
7 bD13
6 bD12
5 bD11
4 bD10
3 bD9
2 bD8
1
U43
DinAEn*
DinBEn*
sDin7
sDin6
sDin5
sDin4
sDin3
sDin2
sDin1
sDin0
AD7 9
AD6 13
AD5 10
AD4 11
AD3 12
AD2 6
AD1 7
AD0 5
R9-R17
A7
A6
A5
A4
A3
A2
A1
A0
Din
Dout
U54
U55
U56
U57
U58
U59
U60
U61
4
RAS0*
15
CAS0*
R20-R21
3
WE1*
14
sDout7
sDout6
sDout5
sDout4
sDout3
sDout2
sDout1
sDout0
BAD7 9
BAD6 8
BAD5 7
BAD4 6
BAD3 5
BAD2 4
BAD1 3
BAD0 2
1
A7 B7
A6 B6
A5 B5
A4 B4
A3 B3
A2 B2
A1 B1
A0 B0
DR E
11 sDout7
12 sDout6
13 sDout5
14 sDout4
15 sDout3
16 sDout2
17 sDout1
18 sDout0
19
DoutAEn*
U44
sDout7 18
sDout6 17
sDout5 14
sDout4 13
sDout3 8
sDout2 7
sDout1 4
sDout0 3
1
D7 O7
D6 O6
D5 O5
D4 O4
D3 O3
D2 O2
D1 O1
D0 O0
OE LE
19 bD15
16 bD14
15 bD13
12 bD12
9 bD11
6 bD10
5 bD9
2 bD8
11
DoutBEn*
U45
AACKB*
U28.5
Schematic 2e
Dual Ported Ram Array
* DRAMs (low byte)



Schematic 3

NMI-Generating Logic



Schematic 4
Tag Register

Parts List and Notes

U1-U6:	74LS245
U7-U8:	74S85
U9:	74LS04
U10:	74LS00
U11:	74S138
U12:	74LS139
U13:	74LS135
U14:	74LS74
U15:	74LS30
U16:	74S03
U17:	74LS32
U20:	74S00
U21:	74S32
U22:	8207-12
U23-U26:	74LS245
U27:	74LS165
U28:	74LS04
U29:	74S138
U30:	74LS08
U31:	74LS32
U32:	74LS138
U33-U34:	74LS243
U35:	74LS74
U36:	74S00
U37:	74LS32
U38-U40:	74LS245
U41:	74LS373
U42-U44:	74LS245
U45:	74LS373
U46-U61:	4164-120
U62:	74LS154
U63:	74LS138
U64:	74LS74
U65:	74LS257
U66-U68:	74LS373
U69:	74LS00
U70:	74LS04
U71:	74LS32
U72:	74LS138
U73-U74:	74LS175
U75:	74LS245
U76-U77:	74LS00

* Note 1: Each TTL IC bypassed by one .1 microfarad ceramic disk capacitor.

* Note 2: Each 4164 bypassed by three .1 microfarad ceramic disk capacitors.

* Note 3: There is a 1K pullup resistor on XRDY inside the Amiga.

Appendix B

Source Code

```
#include "lattice/stdio.h"
#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/gfxmacros.h"
#include "hardware/custom.h"

#define SH_TOGGLES 0
#define SPECIAL 1
#define ASCII 2

      /* Amiga keyboard scan codes */
#define LS_SCAN 0x0060
#define RS_SCAN 0x0061
#define CPLK_SCAN 0x0062
#define CTRL_SCAN 0x0063
#define LALT_SCAN 0x0064
#define RALT_SCAN 0x0065
#define LAMG_SCAN 0x0066
#define RAMG_SCAN 0x0067
#define TAB_SCAN 0x0042
#define I_SCAN 0x0017
#define S_SCAN 0x0021
#define DEL_SCAN 0x0046

      /* Masks for updating status bits in kbd_status */
#define INS_MASK 0x0080
#define CPLK_MASK 0x0040
#define SCROLL_LOCK_MASK 0x0010
#define ALT_MASK 0x0008
#define CTRL_MASK 0x0004
#define LS_MASK 0x0002
#define RS_MASK 0x0001

      /* pixel position of first row, first column in
      display * /
#define ROW0_BASE_LINE 5
#define COL0_LEFT_EDGE 0

      /* height and width of character in pixels */
#define CH_HEIGHT 8
#define CH_WIDTH 8

USHORT kbd_status, r_alt, l_amg, r_amg;
USHORT *afr_dat_ptr, *afr_int_ptr, *kbd_dat_ptr;
USHORT *c_ptr, *treg_clr;
char *treg;
USHORT ticks = 0;
```

```
struct NewScreen mode3_screen =
   {
      0,0,                    /* LeftEdge, TopEdge */
      640,200,                /* Width, Height ( in pixels ) */
      2,                      /* Depth ( number of bit planes ) */
      0,1,                    /* DetailPen, BlockPen */
      HIRES,                  /* ViewMode ( display mode ) */
      CUSTOMSCREEN,           /* Type */
      NULL,                   /* use default font */
      NULL,                   /* DefaultTitle ( screen's title ) */
      NULL,                   /* no screen gadgets */
      NULL                    /* no custom bitmap */
   };

struct NewWindow mode3_window =
   {
      0,0,                    /* LeftEdge, TopEdge */
      640,200,                /* Width, Height */
      0,1,                    /* DetailPen, BlockPen */
      RAWKEY |
      INTUITICKS,          /* IDCMPFlags */
      BORDERLESS |
      SMART_REFRESH |
      ACTIVATE |
      NOCAREREFRESH,
                           /* Flags ( window flags ) */
      NULL,                /* no gadgets */
      NULL,                /* use default checkmark for menus */
      NULL,                /* Title ( window's title ) */
      NULL,                /* Screen ( pointer to screen structure of
                              screen that window will be opened in ) --
                              this value will be filled in after screen is
                              opened */
      NULL,                /* no custom bitmap */
      640,200,
      640,200,
      CUSTOMSCREEN         /* Type ( screen type for this window ) */
   };

struct trans_entry
   {
      USHORT lc;
      USHORT uc;
      USHORT ctrl;
      USHORT alt;
   };

struct trans_entry trans_table[ ] =
   {
      0x0060, 0x007E, 0x007E, 0, 0x0031, 0x0021, 0x0031, 120,
      0x0032, 0x0040, 0x0000, 121, 0x0033, 0x0023, 0x0033, 122,
      0x0034, 0x0024, 0x0024, 123, 0x0035, 0x0025, 0x0025, 124,
```

```
0x0036, 0x005E, 0x001E, 125, 0x0037, 0x0026, 0x0026, 126,
0x0038, 0x002A, 0x002A, 127, 0x0039, 0x0028, 0x0028, 128,
0x0030, 0x0029, 0x0029, 129, 0x002D, 0x005F, 0x001F, 130,
0x003D, 0x002B, 0x002B, 131, 0x005C, 0x007C, 0x001C, 0,
0x0000, 0x0000, 0x0000, 0, 0x0030, 0x0030, 0x0030, 129,
0x0071, 0x0051, 0x0011, 16, 0x0077, 0x0057, 0x0017, 17,
0x0065, 0x0045, 0x0005, 18, 0x0072, 0x0052, 0x0012, 19,
0x0074, 0x0054, 0x0014, 20, 0x0079, 0x0059, 0x0019, 21,
0x0075, 0x0055, 0x0015, 22, 0x0069, 0x0049, 0x0009, 23,
0x006F, 0x004F, 0x000F, 24, 0x0070, 0x0050, 0x0010, 25,
0x005B, 0x007B, 0x001B, 0, 0x005D, 0x007D, 0x001D, 0,
0x0000, 0x0000, 0x0000, 0, 0x0031, 0x0031, 0x0031, 120,
0x0032, 0x0032, 0x0032, 121, 0x0033, 0x0033, 0x0033, 122,
0x0061, 0x0041, 0x0001, 30, 0x0073, 0x0053, 0x0013, 31,
0x0064, 0x0044, 0x0004, 32, 0x0066, 0x0046, 0x0006, 33,
0x0067, 0x0047, 0x0007, 34, 0x0068, 0x0048, 0x0008, 35,
0x006A, 0x004A, 0x000A, 36, 0x006B, 0x004B, 0x000B, 37,
0x006C, 0x004C, 0x000C, 38, 0x003B, 0x003A, 0x003A, 0,
0x0027, 0x0022, 0x0022, 0, 0x0000, 0x0000, 0x0000, 0,
0x0000, 0x0000, 0x0000, 0, 0x0034, 0x0034, 0x0034, 123,
0x0035, 0x0035, 0x0035, 124, 0x0036, 0x0036, 0x0036, 125,
0x0000, 0x0000, 0x0000, 0, 0x007A, 0x005A, 0x001A, 44,
0x0078, 0x0058, 0x0018, 45, 0x0063, 0x0043, 0x0003, 46,
0x0076, 0x0056, 0x0016, 47, 0x0062, 0x0042, 0x0002, 48,
0x006E, 0x004E, 0x000E, 49, 0x006D, 0x004D, 0x000D, 50,
0x002C, 0x003C, 0x003C, 0, 0x002E, 0x003E, 0x003E, 0,
0x002F, 0x003F, 0x003F, 0, 0x0000, 0x0000, 0x0000, 0x0000,
0x002E, 0x002E, 0x002E, 0, 0x0037, 0x0037, 0x0037, 126,
0x0038, 0x0038, 0x0038, 127, 0x0039, 0x0039, 0x0039, 128,
0x0020, 0x0020, 0x0020, 0, 0x0008, 0x0008, 0x0008, 0,
0x0009, 0x0009, 0x0009, 0, 0x000D, 0x000D, 0x000D, 0,
0x000D, 0x000D, 0x000D, 0, 0x001B, 0x001B, 0x001B, 0,
0x007F, 0x007F, 0x007F, 83, 0x0000, 0x0000, 0x0000, 0,
0x0000, 0x0000, 0x0000, 0, 0x0000, 0x0000, 0x0000, 0,
0x002D, 0x002D, 0x002D, 130, 0x0000, 0x0000, 0x0000, 0,
72, 73, 0, 132, 80, 81, 0, 118,
77, 79, 116, 117, 75, 71, 115, 119,
59, 84, 94, 104, 60, 85, 95, 105,
61, 86, 96, 106, 62, 87, 97, 107,
63, 88, 98, 108, 64, 89, 99, 109,
65, 90, 100, 110, 66, 91, 101, 111,
67, 92, 102, 112, 68, 93, 103, 113
};

struct pc_ch
   {
      char c;
      char att;
   };

struct pc_curs
   {
```

```
        char col;
        char row;
    };

struct Screen *pc1_screen, *pc2_screen;
struct Screen *front_screen, *back_screen;
struct Window *pc1_window, *pc2_window;
struct Window *front_window, *back_window;
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct IntuiMessage *message;

main ( )
    {
        if (( IntuitionBase =
           (struct IntuitionBase *)OpenLibrary ( "intuition.library", 1
              ) ) == NULL )
           Cleanup ( "Error: could not open intuition library \n" );
        if ( ( GfxBase =
           (struct GfxBase *)OpenLibrary ( "graphics.library", 1 ) )
              == NULL )
           Cleanup ( "Error: could not open graphics library \n" );

        if ( ( pc1_screen =
           (struct Screen *)OpenScreen ( &mode3_screen ) ) == NULL )
           Cleanup ( "Error: could not open screen \n" );
        mode3_window.Screen = pc1_screen;
        if ( ( pc1_window =
           (struct Window *)OpenWindow ( &mode3_window ) ) == NULL )
           Cleanup ( "Error: could not open window \n" );

        if ( ( pc2_screen =
           (struct Screen *)OpenScreen ( &mode3_screen ) ) == NULL )
           Cleanup ( "Error: could not open screen \n" );
        mode3_window.Screen = pc2_screen;
        if ( ( pc2_window =
           (struct Window *)OpenWindow ( &mode3_window ) ) == NULL )
           Cleanup ( "Error: could not open window \n" );

        kbd_status = r_alt = l_amg = r_amg = 0;
        afr_dat_ptr = (USHORT *)0x200000;
        afr_int_ptr = (USHORT *)0x21E000;
        kbd_dat_ptr = (USHORT *)0x200002;
        treg = (char *)0x21F001;
        treg_clr = (USHORT *)0x21F400;
        front_screen = pc2_screen;
        back_screen = pc1_screen;
        front_window = pc2_window;
        back_window = pc1_window;

        for ( ; ; )
           {
```

```
Wait ( 1 << front_window->UserPort->mp_SigBit );
while ( message = (struct IntuiMessage *)GetMsg (
     front_window->UserPort ) )
   {
      switch ( message->Class )
          {
             case RAWKEY:
                HandleKeys ( );
                break;
             case INTUITICKS:
                ReplyMsg ( message );
                RefVidDisplay ( );
             case CLOSEWINDOW:
                ReplyMsg ( message );
                Cleanup ( NULL );
          }
       }
    }
   }

HandleKeys ( )
   {
      USHORT release;
      USHORT scan_code;

      if ( ( scan_code = message->Code ) > 127 )
         {
            release = 1;
            scan_code = scan_code - 128;
         }
      else
         release = 0;
      ReplyMsg ( message );
      if ( scan_code > 0x005f && scan_code < 0x0068 )
         {
            ShTogKeys ( release, scan_code );
            return ( 0 );
         }
      if ( release )
         return ( 0 );
      if ( l_amg || r_amg )
         AmgKeys ( scan_code );
      else if ( scan_code > 0x004B && scan_code < 0x0050 )
         ArrowKeys ( scan_code );
      else if ( scan_code > 0x0049 && scan_code < 0x005A )
         FuncKeys ( scan_code );
      else if ( kbd_status & ALT_MASK  )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].alt );
      else if ( kbd_status & CTRL_MASK )
         KbdOutput ( ASCII, trans_table[ scan_code ].ctrl );
      else if ( scan_code > 0x0010 && scan_code < 0x001A ||
                scan_code > 0x001F && scan_code < 0x0029 ||
```

```
                    scan_code > 0x0030 && scan_code < 0x0038 )
            AlphaKeys ( scan_code );
         else
            NumPuncKeys ( scan_code );
      }

KbdOutput ( k_class, k_code )
   USHORT k_class, k_code;
   {
      USHORT f_data, k_data;

      f_data = k_data = 0;
      switch ( k_class )
         {
            case SH_TOGGLES:
               k_data = kbd_status;
               f_data = 0x0044;
               break;
            case SPECIAL:
               if ( k_code == 0 )
                  return ( 0 );
               k_data = ( k_code << 8 ) & 0xFF00;
               f_data = 0x0055;
               break;
            case ASCII:
               k_data = k_code & 0x00FF;
               f_data = 0x0055;
         }
         *afr_dat_ptr = f_data;
         *kbd_dat_ptr = k_data;
         *afr_int_ptr = 0;
   }

ShTogKeys ( release, scan_code )
   USHORT release;
   USHORT scan_code;
   {
      switch ( scan_code )
         {
            case CPLK_SCAN:
               shift ( release, CPLK_MASK );
               break;
            case LALT_SCAN:
               shift ( release, ALT_MASK );
               break;
            case RALT_SCAN:
               if ( release )
                  {
                     r_alt = 0;
                     RAltKey ( NULL );
                  }
               else
```

```
            r_alt = 1;
            return ( 0 );
      case CTRL_SCAN:
         shift ( release, CTRL_MASK );
         break;
      case LAMG_SCAN:
         if ( release )
            l_amg = 0;
         else
            l_amg = 1;
         return ( 0 );
      case RAMG_SCAN:
         if ( release )
            r_amg = 0;
         else
            r_amg = 1;
         return ( 0 );
      case LS_SCAN:
         shift ( release, LS_MASK );
         break;
      case RS_SCAN:
         shift ( release, RS_MASK );
      }
   KbdOutput ( SH_TOGGLES, 0 );
}

FuncKeys ( scan_code )
   USHORT scan_code;
   {
      if ( kbd_status & ALT_MASK )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].alt );
      else if ( kbd_status & CTRL_MASK )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].ctrl );
      else if ( kbd_status & LS_MASK  ¦  kbd_status & RS_MASK )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].uc );
      else
         KbdOutput ( SPECIAL, trans_table[ scan_code ].lc );
   }

AmgKeys ( scan_code )
   USHORT scan_code;
   {
      static USHORT ins_tog = 0;
      static USHORT scr_tog = 0;

      switch ( scan_code )
         {
            case TAB_SCAN:
               KbdOutput ( SPECIAL, 15 );
               break;
            case  I_SCAN:
               shift ( ins_tog, INS_MASK );
```

```
            if ( ins_tog )
               ins_tog = 0;
            else
               ins_tog = 1;
            KbdOutput ( SH_TOGGLES, 0 );
            KbdOutput ( SPECIAL, 82 );
            break;
         case  S_SCAN:
            shift ( scr_tog, SCROLL_LOCK_MASK );
            if ( scr_tog )
               scr_tog = 0;
            else
               scr_tog = 1;
            KbdOutput ( SH_TOGGLES, 0 );
            break;
         case DEL_SCAN:
            KbdOutput ( SPECIAL, 83 );
            break;
         case 0x0010:
            Cleanup ( NULL );
      }
   }

ArrowKeys ( scan_code )
   USHORT scan_code;
   {
      if ( ( kbd_status & CTRL_MASK ) && ( kbd_status & ( LS_MASK ¦
           RS_MASK ) ) )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].alt );
      else if ( kbd_status & CTRL_MASK )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].ctrl );
      else if ( kbd_status & ( LS_MASK ¦ RS_MASK ) )
         KbdOutput ( SPECIAL, trans_table[ scan_code ].uc );
      else
         KbdOutput ( SPECIAL, trans_table[ scan_code ].lc );
   }

AlphaKeys ( scan_code )
   USHORT scan_code;
   {
      if ( ( kbd_status & CPLK_MASK ) && ( kbd_status & ( LS_MASK ¦
RS_MASK ) ) )
         KbdOutput ( ASCII, trans_table[ scan_code ].lc );
      else if ( kbd_status & CPLK_MASK )
         KbdOutput ( ASCII, trans_table[ scan_code ].uc );
      else if ( kbd_status & ( LS_MASK ¦ RS_MASK ) )
         KbdOutput ( ASCII, trans_table[ scan_code ].uc );
      else
         KbdOutput ( ASCII, trans_table[ scan_code ].lc );
   }

NumPuncKeys ( scan_code )
```

```
    USHORT scan_code;
    {
       if ( kbd_status & ( LS_MASK ¦ RS_MASK ) )
          KbdOutput ( ASCII, trans_table[ scan_code ].uc );
       else
          KbdOutput ( ASCII, trans_table[ scan_code ].lc );
    }

shift ( release, mask )
   USHORT release, mask;
   {
      if ( release )
         kbd_status = kbd_status & ( ~mask );
      else
         kbd_status = kbd_status ¦ mask;
   }

RefVidDisplay ( )
   {
      struct pc_ch *new_ch;
      struct pc_curs *curs_ptr, curs_pos;
      USHORT row, curr_base_line, t_c;
      char curr_att, tag;
      char txt_buff[ 81 ];
      register USHORT *ch_ptr;
      register USHORT col, i;

      tag = *treg;
      *treg_clr = 0;
      if ( ( tag & 0x01 ) == 0 )
         return ( 0 );
      ch_ptr = (USHORT *)0x210000;
      curs_ptr = (struct pc_curs *)0x211800;
      curs_pos.row = curs_ptr->row;
      curs_pos.col = curs_ptr->col;
      row = curr_att = 0;
      SetDrMd ( back_window->RPort, JAM2 );
      SetAPen ( back_window->RPort, 0 );
      SetBPen ( back_window->RPort, 0 );

      while ( row < 25 )
         {
            curr_base_line = ( row * CH_HEIGHT ) + ROW0_BASE_LINE;
            Move ( back_window->RPort, COL0_LEFT_EDGE,
                 curr_base_line );
            col = i = 0;
            while ( col < 80 )
               {
                  t_c = *ch_ptr;
                  new_ch = &t_c;
                  if ( new_ch->att != curr_att )
                     {
```

```
                        TxtVidOutput ( txt_buff, i );
                        curr_att = new_ch->att;
                        SetAtt ( curr_att );
                        i = 0;
                     }
                  col++;
                  txt_buff[ i ] = new_ch->c;
                  i++;
                  ch_ptr++;
               }
            TxtVidOutput ( txt_buff, i );
            row++;
         }
      SetCursor ( &curs_pos );
      SwitchWindows ( );
   }

TxtVidOutput ( txt_buff, i )
   char *txt_buff;
   register USHORT i;
   {
      if ( i )
         Text ( back_window->RPort, txt_buff, i );
   }

SetAtt ( att )
   char att;
   {
      if ( att & 0x80 )
         SetAPen ( back_window->RPort, 2 );
      else if ( att & 0x08 )
         SetAPen ( back_window->RPort, 3 );
      else
         SetAPen ( back_window->RPort, 1 );
      att = att & 0x77;
      switch ( att )
         {
            case 0x00:
               SetAPen ( back_window->RPort, 0 );
               break;
            case 0x01:
               break;
            case 0x70:
               SetDrMd ( back_window->RPort, JAM2 | INVERSVID );
               break;
            default:
               SetDrMd ( back_window->RPort, JAM2 );
         }
   }

SetCursor ( curs_pos )
   struct pc_curs *curs_pos;
```

```
{
    USHORT x, y;
    char curs;

    curs = 0x20;
    x = (USHORT)( curs_pos->col * CH_WIDTH + COL0_LEFT_EDGE );
    y = (USHORT)( curs_pos->row * CH_HEIGHT + ROW0_BASE_LINE );
    SetDrMd ( back_window->RPort, INVERSVID | COMPLEMENT | JAM1 );
    Move ( back_window->RPort, x, y );
    Text ( back_window->RPort, &curs, 1 );
}

SwitchWindows ( )
{
    struct Screen *t_screen;
    struct Window *t_window;

    t_screen = back_screen;
    t_window = back_window;
    back_screen = front_screen;
    back_window = front_window;
    front_screen = t_screen;
    front_window = t_window;
    ScreenToFront ( front_screen );
    ActivateWindow ( front_window );
}

Cleanup ( exit_text )
char *exit_text;
{
    if ( pc1_window )
        CloseWindow ( pc1_window );
    if ( pc1_screen )
        CloseScreen ( pc1_screen );

    if ( pc2_window )
        CloseWindow ( pc2_window );
    if ( pc2_screen )
        CloseScreen ( pc2_screen );

    if ( GfxBase )
        CloseLibrary ( GfxBase );
    if ( IntuitionBase )
        CloseLibrary ( IntuitionBase );
    if ( exit_text )
        fprintf ( stderr, exit_text );
    printf ( "ticks = %d \n", ticks );
    exit ( !!exit_text );
}
```

```
cr                  equ     0Dh             ; carraige return
lf                  equ     0Ah             ; line feed
bs                  equ     08h             ; backspace
bell                equ     07h             ; bell
        ; destination register of 80186 DMA channel 1
dma1_destreg        equ     0FFD4h
        ; interrupt mask register of 80186 PIC
int_mask_reg        equ     0FF28h
        ; end of interrupt register of 80186 PIC
eoi_reg             equ     0FF22h
        ; PCS6A*
int_enable          equ     1300h
        ; length ( in paragraphs ) of this program
        ; including PSP ( 16 bytes/paragraph )
pgm_len             equ     60h
        ; offset addresses of function number and keyboard
        ; data locations in shared RAM
afr_dat             equ     0000h
kbd_dat             equ     0002h
page0               equ     0A000h
pageA               equ     0A000h
pageB               equ     0B000h
        ; function numbers
kbd_stat            equ     4400h
kbd_key             equ     5500h
        ; addresses of various data in ROM-BIOS data area
kbd_flag1           equ     417h
kbd_flag2           equ     418h
k_buff_start        equ     41Eh
k_buff_end          equ     43Eh
k_buff_head         equ     41Ah
k_buff_tail         equ     41Ch
curr_vid_md         equ     449h
scrn_width          equ     44Ah
regen_len           equ     44Ch
scrn_loc            equ     44Eh
curs_pos            equ     450h
curs_type           equ     460h
disp_pg_num         equ     462h
vid_stat_port       equ     463h
crt_md              equ     465h
        ; copy of cursor position ( in shared RAM )
am_curs_pos         equ     1800h

        ; set interrupt vectors to point to new interrupt
        ; handlers
                    mov     dx, offset NMIint
                    mov     ax, 2502h           ; vector 2
```

```
                int     21h
                mov     dx, offset HardKbdInt
                mov     ax, 250Dh               ; vector 14
                int     21h
                mov     dx, offset SoftKbdInt
                mov     ax, 2516h               ; vector 22
                int     21h
                mov     dx, offset VidInt
                mov     ax, 2510h               ; vector 16
                int     21h

        ; output "interrupt handlers installed" message to
        ; display
                        mov     dx, offset signon
                        mov     ah, 9
                        int     21h

        ; initialize ROM-BIOS data area
                        mov     ax, page0
                        mov     ds, ax
                        mov     byte ptr [ kbd_flag1 ], 0h
                        mov     byte ptr [ kbd_flag2 ], 0h
                        mov     word ptr [ k_buff_head ], 41Eh
                        mov     word ptr [ k_buff_tail ], 41Eh
                        mov     byte ptr [ curr_vid_md ], 07
                        mov     word ptr [ scrn_width ], 80
                        mov     word ptr [ regen_len ], 4000
                        mov     word ptr [ curs_pos ], 0000h
                        mov     word ptr [ curs_type ], 0700h
                        mov     byte ptr [ disp_pg_num ], 0h
                        mov     word ptr [ vid_stat_port ], 3D4h

        ; clear function number and keyboard data locs
                        mov     ax, pageA
                        mov     ds, ax
                        mov     byte ptr [ afr_dat ], 0h
                        mov     byte ptr [ kbd_dat ], 0h

        ; set dest. reg. of DMA channel 1 to point to
        ; address above bottom 1K of I/O address space so
        ; that refresh cycles will not be trapped
        ; NMI-generating logic
                        mov     dx, dma1_destreg
                        mov     ax, 4000h
                        out     dx, ax

        ; disable int. from serial controller, enable INT1
                        mov     dx, int_mask_reg
                        in      ax, dx
                        and     ax, 0FFDFh
                        or      ax, 0010h
                        out     dx, ax
```

```
; enable NMI-generating logic
        mov     dx, int_enable
        out     dx, ax

; terminate and stay resident
        mov     dx, pgm_len
        mov     ax, 3100h
        int     21h

NMIint: proc                    ; NMI handler

        sti
        push    bx
        push    dx
        push    cx
        push    ds
        push    ax
        mov     ax, cs
        mov     ds, ax

    ; read latched address bits
        mov     dx, 1020h
        in      ax, dx
    ; convert address bits to ASCII string, store in
    ; output message
        mov     bx, offset addr_bytes
        call    HexToAscii

    ; read latched status bits
        mov     dx, 1030h
        in      ax, dx
    ; convert status bits to ASCII string, store in
    ; output message
        mov     bx, offset stat_bytes
        call    HexToAscii

    ; get I/O data from AX ( on stack )
        pop     ax
        push    ax
    ; convert I/O data to ASCII string, store in
    ; output message
        mov     bx, offset dat_bytes
        call    HexToAscii

    ; output "I/O port access" message to display
        mov     bx, offset nmi_mess
        call    OutMess

    ; terminate application program
        mov     ax, 4C00h
```

```
                int     21h

NMIint          endp


HardKbdInt      proc      ; keyboard hardware handler

                sti
                push    ax
                push    bx
                push    dx
                push    si
                push    ds
                push    es

                mov     ax, page0
                mov     ds, ax
                mov     ax, pageA
                mov     es, ax

        ; read function number
                mov     ax, es:[ afr_dat ]
        ; is keyboard data a status word?
                cmp     ax, kbd_stat
                je      h_stat
        ; is keyboard data a key code?
                cmp     ax, kbd_key
                je      h_key
        ; unknown function number, exit
                jmp     hkbd_done

        ; read status word from shared RAM
h_stat          mov     ax, es:[ kbd_dat ]
        ; copy status word to keyboard flag byte locs in
        ; ROM-BIOS data area
                mov     byte ptr [ kbd_flag1 ], ah
                and     ah, 0F0h
                mov     byte ptr [ kbd_flag2 ], al
                jmp     hkbd_done

        ; read key code from shared RAM
h_key           mov     ax, es:[ kbd_dat ]
                xchg    ah, al
                cli
        ; set copy of tail pointer ( in bx ) to point to
        ; next key code loc in key buffer
                mov     bx, [ k_buff_tail ]
                mov     si, bx
                add     bx, 2
        ; is copy of tail pointer pointing to end of key
        ; buffer space?
                cmp     bx, k_buff_end
```

```
                jne     hkbd_1
        ; reset copy of tail pointer to point to start of
        ; key buffer space
                mov     bx, k_buff_start
        ; has copy of tail pointer caught up with head
        ; pointer
hkbd_1          cmp     bx, [ k_buff_head ]
        ; if yes, key buffer overflow
                je      hkbd_of
        ; enter key code in key buffer at loc pointed to
        ; by tail pointer
                mov     ds:[ si ], ax
        ; store updated tail pointer in ROM-BIOS data area
                mov     word ptr [ k_buff_tail ], bx
                sti
                jmp     hkbd_done

        ; output "keyboard overflow" message to display
hkbd_of         mov     ax, cs
                mov     ds, ax
                mov     bx, offset kbd_of_mess
                call    OutMess

        ; write specific end of interrupt command to 80186
        ; PIC
hkbd_done       mov     dx, eoi_reg
                mov     ax, 000Dh
                out     dx, ax

        ; clear function number location
                mov     word ptr es:[ afr_dat ], 0h

                pop     es
                pop     ds
                pop     si
                pop     dx
                pop     bx
                pop     ax
                iret

HardKbdInt      endp


SoftKbdInt      proc    ; keyboard I/O handler

                sti
                push    ax
                push    bx
                push    cx
                push    dx
                push    ds
                push    es
```

```
                push    di
                push    si
                push    bp

        ; is service number valid?
                cmp     ah, 3
                jb      s1
                jmp     skbd_done

        ; determine which service is being called
s1              mov     bp, sp
                cmp     ah, 0
                je      ReadChar
                cmp     ah, 1
                je      ReportChar
                cmp     ah, 3
                je      GetShftStat
                jmp     skbd_done

ReadChar        mov     ax, page0
                mov     ds, ax
        ; compare head pointer to tail pointer
rd1             mov     ax, [ k_buff_head ]
                cmp     ax, [ k_buff_tail ]
                jne     rd2
        ; buffer empty, wait for key code to arrive
                jmp     rd1

rd2             cli
        ; read key code pointed to by head pointer
                mov     bx, [ k_buff_head ]
                mov     ax, [ bx ]
        ; return key code in AX stored on stack
                mov     word ptr [ bp ] + 16, ax
        ; set head pointer to point to next key code loc
        ; in key buffer
                add     bx, 2
                cmp     bx, k_buff_end
                jne     rd3
        ; reset head pointer to point to start of key
        ; buffer space
                mov     bx, k_buff_start
        ; store new head pointer in ROM-BIOS data area
rd3             mov     word ptr [ k_buff_head ], bx
                sti
                jmp     skbd_done

ReportChar      mov     ax, page0
                mov     ds, ax
                cli
        ; compare head pointer to tail pointer
                mov     ax, [ k_buff_head ]
```

```
                cmp     ax, [ k_buff_tail ]
                jne     rpt1
        ; buffer empty, set ZF bit in PSW stored on stack
                or      word ptr [ bp ] + 22, 0040h
                jmp     skbd_done
        ; buffer not empty, clear ZF bit in PSW stored on
        ; stack
rpt1            and     word ptr [ bp ] + 22, 0FFBFh
        ; read key code pointed to by head pointer, but do
        ; not update head pointer
                mov     ax, [ k_buff_head ]
        ; return key code in AX stored on stack
                mov     word ptr [ bp ] + 16, ax
                sti
                jmp     skbd_done

GetShftStat     mov     ax, page0
                mov     ds, ax
        ; read keyboard flag byte from ROM-BIOS data area
                mov     al, [ kbd_flag1 ]
        ; return keyboard flag byte in AL stored on stack
                mov     word ptr [ bp ] + 16, ax
                jmp     skbd_done

skbd_done       pop     bp
                pop     si
                pop     di
                pop     es
                pop     ds
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                iret

SoftKbdInt      endp


VidInt          proc    ; video interrupt handler

                sti
                push    ax
                push    bx
                push    cx
                push    dx
                push    ds
                push    es
                push    di
                push    si
                push    bp

        ; is service number valid?
```

```
                cmp     ah, 16
                jb      v1
                jmp     vid_done
        ; only one service currently supported, exit for
        ; all others
v1              cmp     ah, 14
                jne     vid_done

WrTTY
                mov      ax, page0
                mov      ds, ax
                mov      bp, sp
        ; read current cursor position from ROM-BIOS data
        ; area
                mov      cx, [ curs_pos ]
        ; read output character from stack
                mov      ax, [ bp ] + 16

        ; is output character a line feed?
                cmp      al, lf
                jne      wt1
        ; increment row number of cursor position
                inc      ch
        ; if cursor is below the bottom row of display,
        ; need to scroll up rows
                cmp      ch, 24
                jg       wtty_scroll
                jmp      wtty_done

        ; is output character a carriage return?
wt1             cmp      al, cr
                jne      wt2
        ; set column number of cursor position to 0
                mov      cl, 0h
                jmp      wtty_done


        ; simply exit for bell character
wt2             cmp      al, bell
                jne      wt3
                jmp      wtty_done

        ; is output character a backspace?
wt3             cmp      al, bs
                jne      wt4
        ; is cursor on first column of a row?
                cmp      cl, 0
                jg       wt5
        ; is cursor on top row of display?
                cmp      ch, 0
                jg       wt6
        ; cursor on top row, first column; exit
```

```
                jmp     wtty_done
        ; cursor on first column of a row, set cursor
        ; position to last column of row above
wt6             dec     ch
                mov     cl, 79
                jmp     wtty_done
        ; set cursor position left one column
wt5             dec     cl
                jmp     wtty_done

        ; store output character is temporary storage
wt4             mov     cs:[ t_word ], ax
        ; calculate offset word address in video RAM
        ; corresponding to current cursor position
                mov     ax, 160
                mul     ch
                mov     cs:[ row_address ], ax
                mov     ax, 2
                mul     cl
                add     ax, cs:[ row_address ]
                mov     si, ax
                mov     ax, pageB
                mov     ds, ax
        ; read output character from temporary storage
                mov     ax, cs:[ t_word ]
        ; assign character a "normal" attribute
                mov     ah, 03h
        ; write character/attribute pair into video RAM
        ; portion of shared RAM
                mov     ds:[ si ], ax

        ; set cursor position right one column
                inc     cl
        ; is cursor at end of row
                cmp     cl, 80
                jb      wtty_done
        ; set cursor position to first column of row below
                inc     ch
                mov     cl, 0h
                cmp     ch, 25
                jb      wtty_done

wtty_scroll     cld
                mov     ax, pageB
                mov     ds, ax
                mov     es, ax
        ; offset address of data for first column, top row
                mov     di, 0h
        ; offset address of data for first col, second row
                mov     si, di
                add     si, 160
        ; number of character/attribute pairs to move
```

```
                mov     cx, 1920
        ; scroll up one row
          rep     movsw
        ; offset address of data for first col, bottom row
                mov     di, 3840
        ; number of character/attribute pairs
                mov     cx, 80
        ; fill row with "space" character
                mov     ax, 0320h
        ; blank out bottom row
          rep     stosw
        ; set cursor position to first column, bottom row
                mov     cx, 2400h

wtty_done       mov     ax, page0
                mov     ds, ax
        ; store new cursor position in ROM-BIOS data area
                mov     [ curs_pos ], cx
                mov     ax, pageB
                mov     ds, ax
        ; store new cursor position in shared RAM
                mov     [ am_curs_pos ], cx

vid_done        pop     bp
                pop     si
                pop     di
                pop     es
                pop     ds
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                iret

VidInt          endp


HexToAscii      proc    near

                mov     dx, ax
        ; clear second and fourth hex digits ( from left )
                and     ax, 0F0F0h
        ; convert first and third digits into ASCII codes
                mov     cl, 4
                shr     ax, cl
                or      ax, 3030h
        ; store ASCII codes in output message
                mov     [ bx ], ah
                mov     [ bx ] + 2, al
                mov     ax, dx
        ; clear first and third hex digits
                and     ax, 0F0Fh
```

```
        ; convert second and fourth digits into ASCII
        ; codes
                    or      ax, 3030h
        ; store ASCII codes in output message
                    mov     [ bx ] + 1, ah
                    mov     [ bx ] + 3, al
                    ret

HexToAscii          endp


OutMess             proc    near

        ; select "write tty" video service
M1                  mov     ah, 0Eh
        ; read next character in output message
                    mov     al, [ bx ]
                    cmp     al, '$'
        ; if character is '$', terminate
                    je      M2
        ; call video interrupt handler to output character
        ; to display
                    int     10h
                    inc     bx
                    jmp     M1
M2                  ret

OutMess             endp


t_word              dw      ?
row_address         dw      ?
signon              db      'interrupt handlers installed',
cr, lf, '$'
nmi_mess            db      'I/O Port Access', cr, lf
add_mess            db      'Address = '
addr_bytes          db      4 dup(?), cr, lf
stat_mess           db      'Status = '
stat_bytes          db      4 dup(?), cr, lf
dat_mess            db      'Data = '
dat_bytes           db      4 dup(?), cr, lf
term_mess           db      'Program Terminated', cr, lf,
'$'
kbd_of_mess         db      'Keyboard Buffer Overflow', cr,
lf, '$'

end
```

Vita

Name: Robert Thomas Krebs

Born: November 16, 1961 in Dayton, Ohio

Parents: Mr. and Mrs. A.J. Krebs

High School: Middlesex High School (Middlesex, NJ)
graduated in 1980

Undergraduate Study: Bucknell University (Lewisburg, PA)
graduated in 1984 with BSEG degree in
"Computer Science and Engineering"